VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

# Tvorba 3D modelu řidiče na základě hloubkových map

# Creating the 3D Model of Driver from the Depth Map

2016 Bc. Ondřej Fuchsík

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

# Zadání diplomové práce

Student: **Bc. Ondřej Fuchsík**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Tvorba 3D modelu řidiče na základě hloubkových map
Creating the 3D Model of Driver from the Depth Map

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem diplomové práce je vytvořit software pro tvorbu 3D modelu řidiče v osobním automobilu. 3D modelem se rozumí zjištění 3D souřadnic vybraných částí těla (např. hlava, krk, ramena, lokty, ruce) a jejich trajektorií v čase. Hloubková mapa se má vytvářet na základě analýzy hloubkových map;

V diplomové práci proveďte následující:
1. Shrňte dostupnou literaturu a vyzkoušejte dostupné knihovny.
2. Navrhněte vhodné metody detekce jednotlivých vybraných částí řidičova těla.
3. Navrhněte způsob integrace výše uvedených jednotlivých detektorů do systému vytvoření jednoduchého 3D modelu řidičova těla.
4. Navržené řešení realizujte v C/C++.
5. Řešení řádně experimentálně prověřte a porovnejte s řešeními dostupnými.

Postup a výsledky řešení popište v textové části práce. Jako zdroje hloubkových map použijte Microsoft Kinect 2.

Seznam doporučené odborné literatury:

Podle pokynů vedoucího diplomové práce.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **doc. Dr. Ing. Eduard Sojka**

Datum zadání: 01.09.2015

Datum odevzdání: 29.04.2016


| | |
|---|---|
| doc. Dr. Ing. Eduard Sojka | prof. RNDr. Václav Snášel, CSc. |
| *vedoucí katedry* | *děkan fakulty* |

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 29. dubna 2016 .....

Chtěl bych poděkovat vedoucímu práce panu doc. Dr. Ing. Eduardu Sojkovi za cenné a užitečné rady, které vedly k dokončení mé práce. Dále chci poděkovat své rodině a své přítelkyni za každodenní podporu při tvorbě této práce.

**Abstrakt**

Dimplomová práce se zabývá rekonstrukcí modelu řidiče za pomocí RGB-D senzoru Microsoft Kinect druhé generace. Cílem je z nasnímaných dat rozpoznat jednotlivé části lidského těla a ty interpretovat dohromady jako model řidiče. Model řidiče je vypočten na základě analýzy vstupních dat a jednotlivé části těla jsou označeny na základě této analýzy. Analýza zahrnuje detekci torsa, respektive oblasti břicha, ze kterého jsou hledány cesty do jednotlivých končetin a oblasti hlavy. Tyto jsou pak interpretovány jako model řidiče. Navržený analyzátor nepoužívá předtrénovaný klasifikátor poloh a pracuje nad jednotlivými snímky.

**Klíčová slova**: registrace, Kinect, hloubková mapa, mračno bodů, model

**Abstract**

The diploma thesis deals with reconstruction of model of driver from single depth image using RGB-D sensor Microsoft Kinect second generation. The aim of this work is to recognise parts of a human body and interpret those as a model of driver. The model is computed based on analysis of input data and parts of a human body are marked based on this analysis. The analysis includes a detection of a torso or more precisely abdoman area from which paths to limbs and head are searched. These are claimed as a model of driver. The designed analyzer doesn't use pre-trained classifier of poses and it runs over single images.

**Key Words**: registration, Kinect, depth map, point cloud, model

# Obsah

# Seznam použitých zkratek a symbolů

| | | |
|---|---|---|
| CPU | – | Central Processing Unit |
| CUDA | – | Compute Unified Device Architecture |
| FLANN | – | Fast Library for Approximate Nearest Neighbors |
| KDT | – | K-Dimensional Tree |
| MS | – | Microsoft |
| NUI | – | Natural User Interface |
| OUI | – | Organic User Interface |
| PCL | – | Point Cloud Library |
| SDK | – | Software Development Kit |

# Seznam obrázků

# Seznam tabulek

# 1 Úvod

Detekce postavy a částí lidského těla (dále jen „tělo“) z hloubkových map je velice rozšířené téma z oblasti počítačové grafiky a počítačového vidění. Své uplatnění může najít v oblasti herního průmyslu, bezpečnosti, zdravotnictví, automobilového průmyslu, telekonferencí nebo interakcí člověka s počítačem. Stále se ale jedná o výzvu, jelikož lidské tělo je schopné velkého množství poloh a póz.

Velkým milníkem v detekci repektive rekonstrukci póz těla bylo dosaženo v roce 2012 vydáním senzoru *Microsoft Kinect*. Jeho nespornou výhodou je robustnost, rychlost, nízká cena a nepotřebuje žádné pohybové senzory nebo speciální obleky. Toto je vykoupeno poměrně slabou přesností rekonstrukce celého těla v případě, že není objekt zcela vidět a dochází k překryvům. Jeho nástupcem se v roce 2013 stal *Kinect for Xbox One* a jeho verze kompatibilní s operačním systémem Windows vyšla v létě roku 2014 s novou knihovnou SDK 2.0, která již pracuje i na novějších operačních systémech Windows 8 a 8.1. Nýní je možné si koupit jen Kinect pro Xbox One ale existuje oficiální adaptér pro připojení přes rozhraní USB 3.0 s počítačem. Kinect druhé generace má jiné hardwerové specifikace. Zejména se zlepšilo rozlišení barevné kamery na kvalitu 1080p a hloubkové kamery na 512x424 [4].

Cílem mé práce je návrh a implementace softwaru pro automatickou detekci jednotlivých částí těla z hloubkových map, respektive dat pořízených z pobyhového senzoru Microsoft Kinect druhé generace (dále jen „Kinect“). Vstupními daty jsou hloubkové mapy, v případě této práce jsem použil knihovnu PCL viz kapitola 3.4, která ukládá data do formátu PCD jako mračna bodů. Model je vypočten pouze na základě analýzy daného mračna bodů, nevyužívá se tedy žádných vzorových dat nebo strojového učení. Vypočtení modelu je výpočetně náročná operace a jeho rychlost závisí na počtu nasnímaných bodů a rychlosti CPU. Výsledný model reprezentuje lidskou osobu pomocí šesti částí (levá a pravá horní končetina, levá a pravá dolní končetina, hlava a torso).

Práce je rozdělena na několik kapitol. V kapitole 2 se věnuji existujícím metodám, které lze použít pro řešení daného problému. Následuje kapitola 3. Zde se věnuji existujícím knihovnám, pomocí kterých lze řešit některou z částí zadaného problému a zmiňuji zde stručně i dostupnou literaturu k dané problematice. V kapitole 4 jsou uvedeny dvě práce řešící podobnou problematiku ale každá jí řeší jiným způsobem. Jsou zde uvedeny zkrácené postupy, použitá testovací data a dosažené výsledky s těmito daty. V kapitole 8 vyhodnotím dosažené výsledky a také porovnám mé výsledky s výsledky uvedených publikací. Důležitou kapitolou věnující se návrhu řešení je kapitola 5. Popis implementace a detailní popis algoritmů je k nalezení v části 6 Implementace. Následuje návrh experimentů a metrik použitých k možnému porovnání viz kapitola 7. Výsledky experimentů lze nalézt v kapitole 8. Za touto kapitolou se nachází závěr práce.

# 2 Existující metody

## 2.1 Rekonstrukce póz

Rekonstrukce poloh či póz těla z hloubkových dat je rozšířené téma a zabývalo se jím již mnoho nadšenců, odborných pracovníků či komerčních subjektů. Byly vynalezeny různé přístupy k řešení dané problematiky. Obecně bychom mohli rozdělit tyto přístupy na tři skupiny:

1. Přístupy založené na strojovém učení
2. Metody bez znalostí
3. Metody založené na obrazových datech využívající klíčové znaky (siluety, obrysy nebo barvu pleti)

Nejzajímavější jsou metody založené na strojovém učení a bezznalostní metody, jelikož s příchodem Kinectu a možnosti použití hloubkových informací, dovolilo řešit problémy metod založených na obrazových datech.

### 2.1.1 Přístupy založené na strojovém učení

Obecně přístupy založené na strojovém učení, závisí na tom jak je natrénován klasifikátor. Z tohoto důvodu všechny metody používající strojové učení potřebují velké množství trénovacích dat. Obvykle jsou tato data generována automaticky a obsahují syntetická hloubková data lidí, různých velikostí a postav ve velice různorodých pozicích. Těmito daty jsou pak trénovány například náhodné rozhodovací lesy (randomized decision forests).

Aby tyto metody mohly dobře fungovat, potřebují stovky tisíc trénovacích dat. Klasifikátor je často urychlován tak, že se spustí pro každý pixel paralelně na grafické kartě. Samotný proces pak zjednodušeně probíhá tak, že se nasnímá jeden rámec dat za druhým, a pro každý se pomocí klasifikátoru určí, která póza z databáze odpovídá póze nasnímané.

### 2.1.2 Metody bez znalostí

Přístupy spadající do této kategorie jsou odlišné od předchozí zejména tím, že nevyužívají strojového učení a nemají tedy žádnou vstupní databázi testovacích dat a nezávisí u nich tedy na kvalitě testovacích dat, tak jako u předchozí skupiny, kde byla jednou z kritických fází, fáze trénovaní klasifikátoru.

Naopak, tyto metody často využívají okolností v jakém prostředí a s jakými daty budou pracovat. Jsou tedy většinou omezeny svým použitím, pro které ale pracují velice dobře a efektivně. Jsou postaveny na určitém matematickém aparátu nebo myšlence, pomocí které jsou schopny najít, rozeznat a označit jednotlivé části těla. Jedná se tedy o zakódování určitého modelu, do programovacího jazyka.

## 2.2 Metody nalezení nejkratších cest

V této podkapitole jsou zmíněny existující metody nalezení nejkratších cest. Většina z těchto metod je dostupná i v různých C++ knihovnách. První část je věnována obecnému vysvětlení problému nalezení nejkratší cesty a je zde tento problém i formálně definován.

### 2.2.1 Definice

*Cesta* je sekvence vrcholů $\langle v_0, v_1, ..., v_k \rangle$ v grafu $G = (V, E)$, kde každá hrana $(v_i, v_{i+1})$ je v množině $E$. Dále je definována váha $w(u, v)$ pro každou hranu $(u, v)$. *Váha cesty (délka cesty)*, je suma vah každé hrany v cestě:

$$w(p) = \sum_{i=0}^{k-1} w(v_i, v_{i+1})$$

*Nejkratší délka cesty* z vrcholu $u$ do $v$ je minimum ze všech možných délek cest:

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \rightsquigarrow v\} & \text{pokud existuje cesta z } u \text{ do } v \\ \infty & \text{jinak} \end{cases}$$

*Nejkratší cesta* je kterákoliv cesta, jejíž váha je rovna nejkratší délce cesty [7]. Řešení této úlohy spadá do kategorie NP-úplných problémů z teorie grafů. Jeho základní překážkou je existence cyklů záporné délky, poté pak není možné řešit tuto úlohu v polynomiálním čase. Nelze díky těmto cyklům říci, jestli již byla nalezena nejkratší cesta pro daný uzel. Obecně ale úlohy, které ústí v použití tohoto algoritmu záporné cykly neobsahují již z principu. Například hledáme-li na mapě nejkratší cestu mezi městy, tak jednotlivé vzdálenosti mezi městy nikdy zápornou délku mít nebudou.

Všechny níže uvedené algoritmy pracují na *Bellmanovu principu optimality*, který říká, že pokud graf neobsahuje cyklus záporné délky, tak pto každé tři vrcholy $x, y, z$ platí:

$$u(x, y) = min_{x \neq y}(u(x, z) + a(z, y))$$

Kde $u(x, y)$ reprezentuje délku cesty z vrcholu $x$ do vrcholu $y$ a $a(z, y)$ reprezentuje vzdálenost vrcholu $z$ od vrcholu $y$. Z této rovnice vyplývá, že každá nejkratší cesta se opět skládá pouze z nejkratších cest.

### 2.2.2 Varianty algoritmů

Existují různé varianty hledání nejkratších cest [7]. *Jednopárová varianta nalezení nejkratší cesty (single-pair shortest-path)* řeší problém nalezení nejkratší cesty mezi jedním párem vrcholů. *Nalezení nekratší cesty s jedním zdrojem (single-source shortest-path)* je varianta, ve které je potřeba nalézt nejkratší cestu z jednoho vrcholu do všech ostatních vrcholů v grafu. Takto získaným cestám se říká *strom nejkratších cest (shortest-path tree)*. Jako poslední se v literatuře [2] uvádí varianta *nalezení nejkratší cesty pro všechny dvojice (all-pairs shortest-path)*. Existují různé algoritmy řešící daný problém. Uvedu zde algoritmy, které řeší vždy nějakou variantu z výše uvedených a stručně popíši, kdy je nejlepší určitý algoritmus použít.

*Dijkstrův algoritmus* řeší variantu s jedním zdrojovým uzlem (vrcholem), s ohodnocenými hranami. Hrany mohou být orientované i neorientované, ale jednotlivé hrany nesmí mít záporné hodnoty. V případě, že jsou některé hrany ohodnoceny zápornými hodnotami, je lepší použít Bellman–Fordův algoritmus. Pokud jsou hrany ohodnoceny pouze váhami s hodnotou jedna, je vhodné použít variantu Dijsktrova algoritmu, který prohledává do šířky. Asymptotická složitost algoritmu je $O(|H| \cdot \log |U|)$.

*Bellman–Fordův algoritmus* řeší variantu s jedním zdrojovým uzlem a není závislý na tom, zda jsou hrany ohodnoceny kladnými, či zápornými hodnotami. Pokud jsou však ohodnoceny všechny hrany kladnými hodnotami, podává lepší výsledky algoritmus Dijkstrův. Jeho asymptotická složitost je $O(|H| \cdot |U|)$

*Johnsonův algortimus* je zaměřen na hledání nejkratších cest mezi všemi dvojicemi vrcholů v grafu. Jedná se o časově náročný algoritmus jehož časová složitost je $O(|U||H| \log |U|)$.

Algoritmy řešící nalezení nejkratší cesty lze použít v mnoha oblastech a mají široké využití. Jedním z nich je například ve směrování packetů nebo také v počítačové grafice, o čemž jsem se přesvědčil během psaní této práce.

# 3 Dostupná literatura a knihovny

## 3.1 Kinect SDK

K dispozici je několik knih o programování s Kinectem, ať už s jeho druhou generací nebo první. Jelikož je tato práce zaměřená spíše na novější verzi Kinectu musím zmínit knihu *Beginning Kinect Programming with the Microsoft Kinect SDK*[17]. Tato publikace obsahuje vše potřebné k tomu, aby si čtenář byl schopen Kinect úspěšně nainstalovat, propojit s počítačem a operačním systémem Windows. Je podrobně popsáno jak pracovat se senzorem Kinectu stejně tak jako s proudem obrazových dat. Celá jedna kapitola je věnována práci s hloubkovými daty, od měření hloubky přes vylepšené hlubokové mapy až po zpracování obrazu. Velice zajímavé a přínosné jsou kapitoly o *Skeleton Trackingu*, kde v základní části autoři vysvětlují samotný objektový model *Skeletonu*, prostorové transformace a ovládání *SkeletonViewer*. V rozšířené části se pak zaobírají uživatelskými interakcemi. Předposlední kapitola je zaměřena na gesta a různé implementace jak gesta detekovat. Probrány jsou obecně známá gesta, jako je například přesunutí, stisknutí tlačítka nebo skrolování. Poslední kapitola je věnována zvuku a ovládání mikrofonu Kinectu.

## 3.2 Hloubkové mapy a algoritmy

Pro přiblížení hloubkových map zmíním knihu *Depth Map and 3D Imaging Applications*[18]. Obsahuje velké množství technik a algoritmů v souvislosti s 3D snímáním, zobrazováním, rekonstrukcí obrazu, digitalizací objektů, robotického vidění, rekonstrukcí póz či rozeznání 3D obličeje. Jednotlivé části jsou dostupné v podobě odborných článků. V souvislosti s touto diplomovou prací je velice zajímavá kapitola 28, která se zabývá obnovou 3D pózy těla z hloubkových map, při použití siluet nasnímané postavy a strojového učení.

## 3.3 OpenNI

*OpenNI (Open Natural Interaction)* je open source software, vytvořený skupinou neziskových organizací v čele se společností *PrimeSense* s cílem vytvořit API pro *middleware komponenty* a *Natural Interaction Devices* tj. zařízení zachycující zvuk a pohyb těla pro přirozenější práci uživatele s počítačem. Příkladem takového zařízení je právě Kinect. *Middleware komponenty* popisuje *OpenNI* jako softwarové jednotky umožňující přidělení sémantiky obrazových, či zvukových dat a jejich analýzu. Právě společnost *PrimeSense* dodala komponentu *NITE*, která zpřístupňuje senzory Kinectu a poskytuje algoritmy pro sledování těla. Obecné API poskytuje podporu pro hlas a rozeznávání hlasových příkazů, gesta a také sledování pohybu těla. Projekt *OpenNI* je od dubna 2014 pozastaven.

## 3.4 Point Cloud Library

Jak již název napovídá, jedná se o knihovnu pro práci s mračny bodů. Mračno je definováno jako datová struktura, reprezentující množinu obecně $n$-dimenzionálních bodů, obvykle však uchováváme dimenze tři (pro každou souřadnou osu jednu), respektive čtyři pokud chceme uchovat i barvu bodu. Mračna je možné pořizovat ze senzorů jako jsou stereo kamery, 3D skenery nebo pomocí OpenNI API. To nám umožňuje načítat hloubková data z Kinectu přímo do tohoto datového typu, se kterým lze dále pracovat jako s jinými datovými typy.

Celá knihovna (framework) obsahuje celou řadu algoritmů pro filtrování, výpočet klíčových bodů, rekonstrukci povrchu, registraci a další užitečné funkce pro práci s mračny. Knihovna je zdarma pro komerční a vědecké použití, je multiplatformní a byla úspěšně zkompilována a použita na platformě Linux, MacOS, Windows a Android/iOS. Knihovna je vyvíjena velkým počtem vývojářů,inženýrů a vědců z různých organizací a zeměpisných šířek. Pro příklad: nVIDIA, TOYOTA, Universität Osnabrück, National Institut of Standards and Technology, Texas A&M University a mnoho dalších.

## 3.5 Boost Graph Library

Tato knihovna zpřístupňuje grafové struktury a algoritmy nad těmito strukturami, ale navenek skrývá jejich implementaci. Její silnou stránkou je možnost substituovat typy jako hrana a uzel, za své vlastní a obecně si za používané datové typy knihovny dosadit vlastní datové typy, či celé struktury. Poskytované algoritmy tak lze spouštět nad svými strukturami. Velkou nevýhodou této knihovny je její neintuitivnost, složitost použití a programátorsky přívětivá není ani dokumentace, které chybí zejména stylizace, ale hlavně příklady použití. Docela často je programátor nucet hledat dlouze různá řešení na internetu a ty si podle svého programátorského umu, přizpůsobit aktuálním potřebám. Na webových stránkých knihovny jsem našel referenci na knihu [2], která je uživatelským manuálem a referenčním manuálem knihovny. Kniha slouží jako kompletní průvodce knihovnou a tutoriálem v jednom.

V knihovně PCL, která je v rámci práce také využívána, se nenachází žádný modul pro práci s body z PCL knihovny jako s uzly v grafu. Jediná zmínka, je v prezentaci v rámci PCL tutoriálů viz [14]. V této prezentaci zmiňují modul graphs v knihovně pcl, ale zatím jako experimentální. Není zahrnut v dostupném kódu této knihovny, navíc není označen za stabilní. Cílem tohoto modulu je umožnit práci s mračnem jakoby se jednalo o graf, kde jednotlivé body, jsou vrcholy grafu. Vzdálenosti mezi body jsou váhami hran. Následně by měl tento modul umožnit běh grafových algoritmů nad mračnem, bez nutnosti komponovat vlastní řešení.

### 3.6 Persistence1D

Jedná se o volně přístupnou a volně použitelnou knihovnu pro zjišťování minimálních a maximálních hodnot v souboru dat. Knihovna umožňuje extrahovat jak lokální tak globální extrémy funkcí. Kód této knihovny pracuje s asymptotickou složitostí $O(n \cdot \log n)$. Více lze nalézt na stránkách knihovny [3].

# 4 Existující řešení

V této kapitole bych rád uvedl existující řešení rozpoznání poloh lidského těla v hloubkové mapě. Uvedu dvě existující práce, nastíním zbůsob jejich řešení a uvedu jejich výsledky. Zde uvedené poté poslouží k závěrečnému porovnání s mým řešením. Jednotlivé práce jsem se snažil vybrat tak, aby postup nebo styl použití byl relativně podobný s možným použitím této publikace. Vybíral jsem jednotlivé publikace i s návazností na předchozí kapitolu 2.

## 4.1 Strojové učení a rozhodovací stromy

První publikací je [0] , která používá k rozpoznání poloh vždy jeden snímek hloubkové mapy k rozpoznání polohy a to bez dočasných informací. Testovací data získávali z *motion capture* systému a také generováním reálných syntetických dat lidí v mnoha pózách a velikostí. Tato data pak sloužila k natrénování klasifikátoru a sestrojení *random decision trees*. Použití rozhodovacích stromů s trénovácí sadou několika set tisíc snímků jim dovolilo řešit překryvy.

Jejich zjednodušený postup řešení je následující:

1. Objekt v hloubkové mapě je rozdělen na části (tento krok je založen na porovnávání na úrovní pixelů (*per-pixel problem*)). Definují tedy několik částí v určité lokaci, které hustě pokrývají daný objekt. Některé části slouží pouze jako výplň, jiné reprezentují právě uzly hledané kostry *(skeletal joints)*.

2. Tyto části jsou specifikovány v texturovací mapě. Dvojice hloubkové informace a části těla jsou potom vstupy do trénovacího klasifikátoru. Pro jednotlivé části jsou vytvořeny spojovací uzly (přes rozhodovací stromy)

Pro testování systémů použili jak reálná data, tak i synteticky vygenerovaná data, kde syntetických snímků použili 5000 a reálných 8808. Použili 15 různých osob s různým oblečením. Dále použili data z práce [16]. Pro běh experimentů měli nastavené limity rotace osoby od -120 stupňů do +120 stupňů, protože je systém navrhován pro prostředí, kde je osoba otočena čelem ke kameře. Jejich hlavní metrikou je měření přesnosti vyhodnocení jednotlivých částí těla. Jednotlivé metriky pro každou část mají svou váhu podle velikosti dané části. Jejich systém je schopen zpracovat 200 snímků za sekundu. Mnoho operací je urychlováno na GPU jelikož hlavní platformou je herní konzole Xbox 360. Na moderním 8 jádrovém procesoru zpracuje jejich systém 50 snímků za vteřinu. Výsledky přesnosti z důvodů rozsáhlosti neuvádím, ale lze je nalézt v uvedené publikaci.

## 4.2 Geodetické vzdálenosti v kombinaci s registrací mračen bodů

Druhou publikací je [1], která odhaduje polohy lidského těla na základě geodetických vzdáleností a registrace mračen. V této publikaci je použito grafové reprezentace mračna bodů, to umožňuje měřit geodetické vzdálenosti nezávisle na poloze lidského těla. To dělá tento systém robustní. Popsaná metoda nepotřebuje předtrénovaná data a ani inicializační polohu objektu v mračně bodů. Jejich metoda má následující části:

1. Načtení hloubkové mapy jako 3D mračno bodů.
2. Detekce trupu. Střed trupu je detekován na základě vzdálenostní transformace [5]. Potom je bod označen za část trupu, pokud jeho vzdálenostní transformace je menší než polovina vzdálenostní transformace obrazu.
3. Vypočtení geodetických vzdáleností a cest. Geodetické vzdálenosti počítají od centra trupu do všech ostatních bodů. Z cest je vytvořen graf, kde jednotlivé vrcholy jsou body mračna a jsou spojeny hranami pokud platí určitá pravidla (kritéria).
4. Označení geodetických cest.
5. Segmentace těla na části.
6. Přizpůsobení spojům kostry.

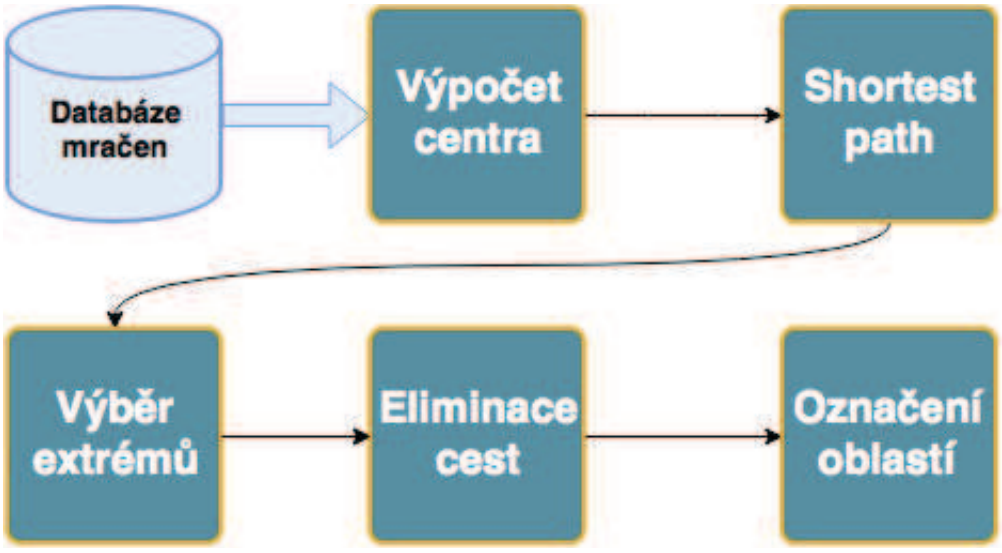
K testování metody využili syntetické hloubkové mapy, které byly generovány z animovaného 3D objektu v OpenGL. Tento způsob má dle této publikace výhodu, že jsou známy přesné polohy spojů v kostře a mohou být porovnány s výsledky. Použili přibližně 25000 obrázků. Osoba na obrázcích je otočena směrem ke kameře s maximální rotací +-40 stupňů. Pro každý obrázek je spočtena euklidovská vzdálenostní chyba pro každý spoj kostry. Střední odchylka byla 2cm a maximální chyba byla 10cm. Někdy se chyba vyšplhala až na 28cm, když byly ruce přímo proti kameře. V takových situacích je potom málo bodů k dobré detekci. Dále uvádějí průměrné chyby pro jednotlivé části těla. Pro hlavu je průměrná chyba 0.6cm, pro chodidla a lokty 3.8cm. Systém zpracovává 20 snímků za vteřinu.

# 5 Návrh řešení

Téma diplomové práce je *Tvorba 3D modelu řidiče na základě hloubkových map*, jehož stěžejní částí je označit v hloubkové mapě jednotlivé části těla (ruce, nohy, hlavu a trup) (viz obr. 2/s22), popřípadě významné body těla (lokty, ramena, krk, hrudník a kolena). Jedná se o netriviální problém z oblasti počítačového vidění a zpracování obrazu, které je možné řešit mnoha způsoby. Takový způsob je nutné vybírat zejména na základě vstupních dat, známých omezení např. prostředí, technických limitů a požadavků na kvalitu či přesnost. Otázka tedy zní: Jak docílit označení vyjmenovaných částí v hloubkové mapě?

Obecně jsou dva přístupy jak daný problém řešit (popsáno v kapitole 2). Jelikož nemám k dispozici obrovskou množinu syntetických dat a vzhledem k budoucímu možnému použití některých částí této práce jsem se rozhodl vyřešit tuto otázku pomocí bezznalostních metod. V následujích podkapitolách je popis mé metody, která vznikla na základě studování existujících řešení podobných próblémů, vlastních úvah a znalostí, konzultací s vedoucím mé diplomové práce a informacích nabytých při studování uvedených zdrojů.

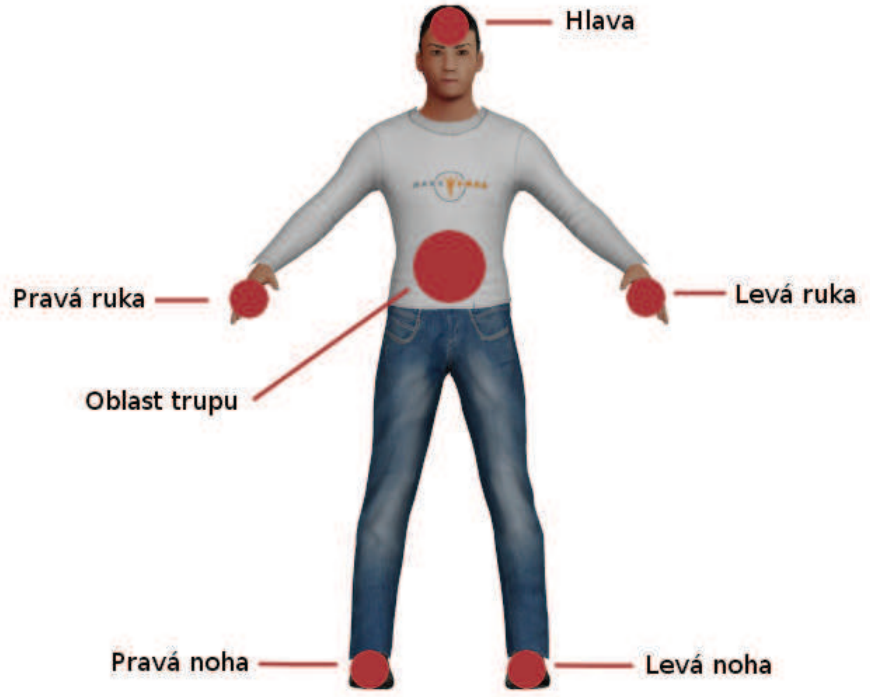
Při návrhu jsem použil obecnou metodu zdola nahoru, protože jsem postupoval od jednodušších částí ke složitějším, tyto části se pak dají využít i samostatně (viz obr. 1/s21). Jednotlivé bloky jsem si poté rozděloval na podproblémy pomocí návrhu shora dolů. Dělení na podkapitoly je založeno na výstupech z metody zdola nahoru.




Obrázek 1: Návrh vlastní metody

Do návrhu jsem předem zahrnul i použití některých volně dostupných knihoven. Jelikož bude použit jako vstupní médium Kinect, a knihovna PCL nabízí možnost načtení a uložení dat z Kinectu na disk. Rozhodl jsem se ji použít jako hlavní zdroj pro práci s hloubkovými mapami. Vstupní data budou načítána pro testovací účely z disku. Po načtení snímku z datového úložiště nebo z Kinectu budou na hloubkové mapě spuštěny operace pro identifikaci jednotlivých částí těla.

Dále rozebíraný návrh předpokládá splnění prerekvizit uvedených v podkapitole 5.1. Nejdříve se nalezne oblast trupu, respektive břicha viz podkapitola 5.2 a poté se z tohoto bodu budou hledat cesty do jednotlivých končetin a oblasti hlavy. K tomuto jsem se rozhodl použít knihovnu BGL, protože hledání nejkratší cesty v grafu je jeden ze známých problémů z teorie grafů viz podkapitola 5.5. Aby bylo možné použít metody pro výpočet nejkratších cest k ostatním bodům z této knihovny, musí se naplnit datové struktury této knihovny již existujícími daty, nebo převést jejich reprezentaci do reprezentace, kterou dokáže tato knihovna použít. Zejména se jedná o základní třídu grafu. Toto bude popsáno v podkapitole 5.4, kde bude využita i knihovna PCL a struktura KD-tree viz podkapitola 5.3. Z knihovny BGL bude použit algoritmus pro výpočet nejkratších cest a věnována je mu podkapitola 5.5. Následují podkapitoly 5.6, 5.7 a 5.8 věnující se výběru správných cest a označení částí těla.




Obrázek 2: Vymezení základních částí k označení.

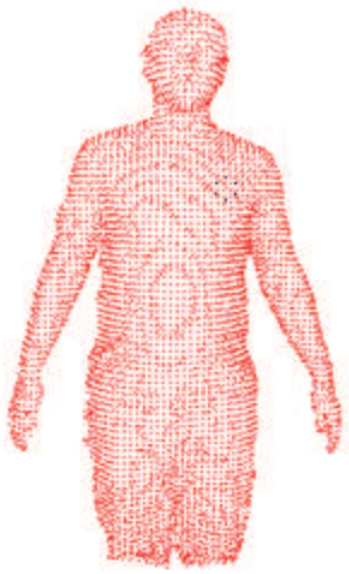
## 5.1 Prerekvizity

Před samotným popisem řešení je nutné zmínit, že aplikace spoléhá na obsah vstupních dat. Základním problémem, který tento návrh řeší je označení jednotlivých částí těla. Aplikace tedy očekává vstupní data, obsahující nasnímanou postavu Kinectem v určitém prostředí. Pro zjednodušení a zaměření se na hlavní úkol, je předpokládáno, že daná osoba stojí před konstantním pozadím. To umožní v aplikaci jednoduše zahodit body, které nejsou potřebné pro řešení problému, na základě hloubkové informace.

## 5.2 Výpočet centra

V této části bude popsán proces jakým se zjišťuje oblast břicha respektive trupu (viz obr. 2/s22). Vstupem je tedy PCD soubor uložený na disku, ten je dále načten do paměti. Nejdříve se na základě hloubky (osa Z) zahodí všechny body za mezí danou uživatelem. Tato mez se zadává ručně, bližší popis lze najít v kapitole 6.

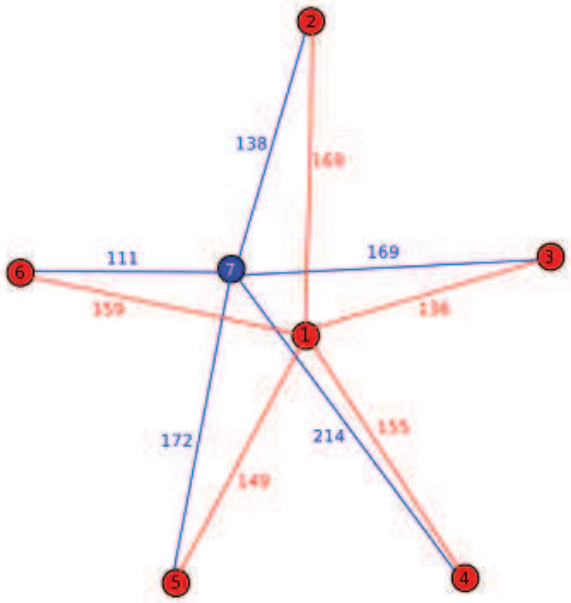
Následně se mračno převzorkuje přes voxelovou mříž. Voxel je částice objemu v pravidelné mřížce třídimenzionálního prostoru [12]. Velikost jednoho voxelu, je nastavena standardně na 1.5 cm ale je možné ji zvětšit, či zmenšit pomocí parametrů aplikace viz kapitola 6. V rámci voxelu, je spočteno težiště mezi body, které do daného voxelu spadají. Tímto těžištěm jsou pak nahrazeny všechny body v daném voxelu. Převzorkováním se výrazně urychlí následující výpočty a to za minimální ztrátu přesnosti, za předpokladu, že je vhodně zvolená velikost jednoho voxelu. Obecně má mračno s nasnímanou osobou okolo 32000 bodů, převzorkované mračno se standardní velikostí voxelu má přibližně 6000 bodů.



Obrázek 3: Ukázka osoby v PCD formátu.

Následuje proces hledání centra, či trupu. Pokud byly dodrženy prerekvizity, správně nastaven limit osy Z a ponechána základní velikost voxelu, či nastavena na jinou vhodnou hodnotu, měly by být k dispozici body reprezentující osobu (viz obr. 3/s23). V takové množně dat, potřebujeme najít bod, který se nachází někde v oblasti trupu, například v oblasti břicha.

Vzhledem k různorodosti pozic, v jakých se může osoba nacházet, je nutné najít nějaký obecný přístup k řešení tohoto úkolu. Podíváme-li se podrobněji na hledaný bod, zjistíme, že vzhledem k ostatním bodům mračna má v součtu nejkratší vzdálenost. Vycházel jsem z obrázku (viz obr. 4/s24), který znázorňuje velmi zjednodušenou osobu v šeti bodech (kolečka s číslem uprostřed). Jednotlivé body reprezentují konce končetin (3, 4, 5, 6), hlavu (2) a náš hledaný bod č. 1. V obrázku jsou naznačeny vzdálenosti od bodu č. 1 k ostatním bodůmm (červená barva). Součet těchto vzdáleností je roven 768px. Dále je v obrázku zakreslen bod s č. 7. Pro tento bod jsou zakresleny také vzdálenosti k ostatním bodům (modrá barva). Součet těchto vzdáleností je roven 804px. Lze vidět, že bod blíže středu pomyslného trupu (břicha), má menší součet vzdáleností. Vzdálenosti jsou změřeny v pixelech v grafickém editoru Inkscape.




Obrázek 4: Zjednodušený model osoby.

Aplikace tedy projde všechny body načteného mračna a pro každý bod spočte euklidovské vzdálenosti k ostatním bodům. Vypočtené vzdálenosti se sečtou a výsledek se uloží k danému bodu a pokračuje s dalšími body v řadě. Po spočtení součtů pro všechny body vybere nejmenší z výsledných součtů. Například z výše uvedeného příkladu by vybral součet u bodu č. 1 a za referenční bod v oblasti břicha by byl prohlášen bod s č. 1.

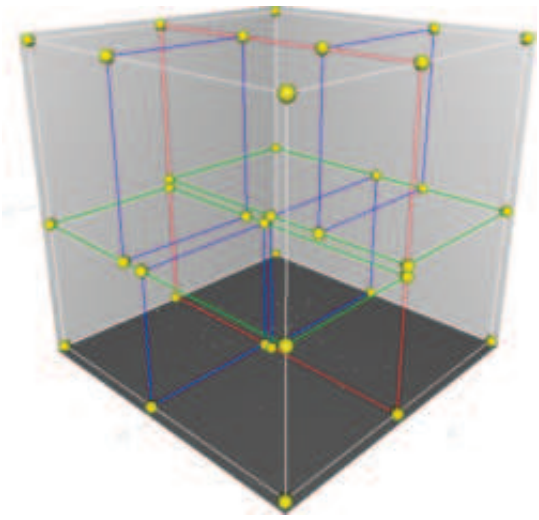
## 5.3 FLANN KDTree

Tato podkapitola je věnována struktuře KD-tree a hledání nejbližších sousedních bodů, které jsou použity více krát v této práci. Slouží také jako doplnění představy o použitých algoritmech v této práci.

Knihovna PCL obsahuje implementaci FLANN KD-tree nad určitým mračnem bodů. Dokáže nalézt nejbližší sousedy na základě:

1. Vzdálenosti od daného bodu
2. Počtu nejbližších bodů od daného bodu

KD-tree (tj. k-dimensionální strom) je datová struktura používaná k organizování množiny bodů v k-dimensionálním prostoru [6]. Je velmi užitečná pro vyhledávání nejbližších sousedních bodů. V knihovně PCL jsou tyto stromy výhradně třídimenzionální. Během dělení prostoru na podprostory se vytváří binární strom. Celý proces dělení je nastaven tak, aby v každé buňce ležel určitý přibližně stejný počet bodů, která při dělení prostoru vznikne. Na každé úrovni jsou potomci rozdělení pomocí roviny, která je kolmá k příslušné ose. V kořenovém uzlu stromu jsou potomci dělení podle osy X. V každé další úrovni se potomci dělí na základě následujících os. V momentě, že se vyčerpají osy, se začíná opět od osy X. K určení dělícího bodu, ke kterému je sestrojena kolmá rovina, se používá medián. Ukázka 3D stromu je znázorněna na obrázku 5. Celý proces hledání nejbližších sousedních bodů začíná sestrojením KD-stromu. Poté co je dosazen bod a vzdálenost v jaké se mají hledat nejbližší body, se začne rekurzívně procházet strom a hledá se nejlepší shoda.



Obrázek 5: Ukázka třídimensionálního kd-tree [6]

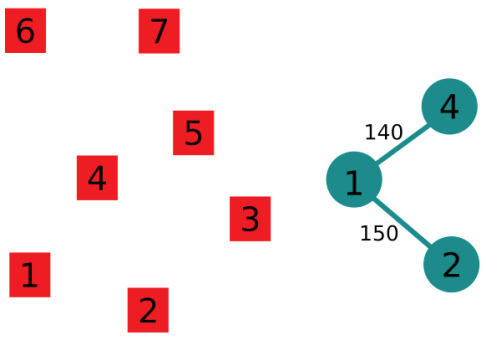
## 5.4 Převod z PCL do boost graph library

V této podkapitole je popsáno vytváření grafu, který bude použit v algoritmu 2.2.2 z BGL v podkapitole 5.5. Základním problémem, je nějakým způsobem vytvořit graf, který bude možné použít ve zmíněné knihovně. Základními stavebními prvky grafu jsou uzly, hrany a ohodnocení hran. Uzel může být v podstatě jakákoliv struktura, ale musí obsahovat číselný index, podle kterého je BGL schopna se na daný uzel odkazovat. Pro hrany je možné opět použít vlastní struktury, ale protože potřebujeme uchovávat vzdálenosti mezi dvěmi body (uzly), je nutné v hraně mít číselnou položku v pohyblivé desetinné tečce.

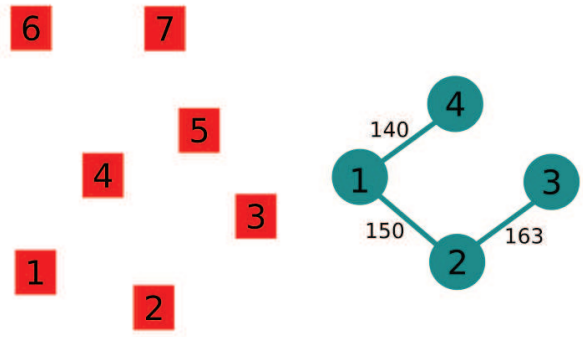
Otázkou je: Jak naplnit strukturu grafu? Mé řešení lze shrnout do následujících bodů:

1. Vytvořit strukturu grafu, která bude mít velikost počtu bodů v aktuálním mračnu (kolik bodů, tolik uzlů).
2. Vybrat první bod z mračna.
3. Nalézt N nejbližších bodů k tomuto bodu.
4. Přidat hrany do grafu, mezi tímto bodem a nalezenými body (jen pokud hrana neexistuje).
5. Načíst další bod a začít znovu od bodu 3, pokud se nejedná o bod poslední.

Tímto docílíme sestrojení grafu na základě existujicího mračna. Na jednotlivé body se budeme moci odkazovat pomocí indexů, jak ve struktuře pro BGL, tak ve struktuře *pcl::PointCloud* v mračnu. Během přidávání hrany do grafu, je nutné kontrolovat, zda již hrana v grafu neexistuje. Pokud ano, danou hranu nebudeme duplikovat. Jakmile víme, že můžeme hranu přidat do grafu, uložíme s ní také vzdálenost mezi body, které tato hrana spojuje. Tato vzdálenost je automaticky počítáná při hledání nejbližších sousedních bodů.



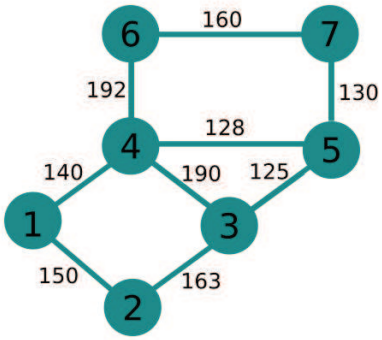

Obrázek 6: Ukázka první iterace metody.




Obrázek 7: Ukázka druhé iterace metody.

Na obrázcích 6 a 7, jsou zobrazeny dva kroky této metody. V prvním kroku jsou pro bod 1 nalezeny nebližší body 2 a 4. Pro tyto dvojice jsou přidány hrany se vzdálenostmi do grafu. V druhém kroku jsou pro bod 2 nalezeny nebližší body 1 a 3. Přidána je hrana pouze pro body

2 a 3, protože druhá hrana již v grafu existuje. Ukázkové mračno je reprezentováno červenými a očíslovanými čtverci. K němu je komplementárně tvořen graf dle výše popsaného návrhu řešení. Uzly grafu jsou vyobrazeny jako kolečka, mezi nimi jsou neorientované hrany se vzdáleností a uzly obsahují index bodu. V každém kroku se hledají nejbližší dva body. Začíná se v bodě 1, jehož nejbližší dva body jsou 2 a 4. V grafu se tedy přidají hrany mezi body 1, 2 a mezi body 1, 4 (viz obr. 6/s26) . V dalším kroku se hledají dva nejbližší body pro bod 2. Těmito body jsou 1 a 3. Přidání hrany mezi body 2 a 1 se přeskočí, protože pro ně již v grafu existuje hrana. Pro body 2 a 3 se do grafu přidá hrana (viz obr. 7/s26). Takto by se pokračovalo až do té doby, než by se zpracoval poslední bod 7. Výsledný graf (viz obr. 8/s27).




Obrázek 8: Finální graf po provedení celé metody.

## 5.5 Dijkstrův algoritmus nejkratší cesty

Hlavní vstupní komponentou do této části návrhu je graf vytvořený v předchozí podkapitole 5.4 z analyzovaného mračna. Cílem této části bude najít nejkratší cesty z jednoho vrcholu grafu do ostatních vrcholů. Počátečním vrcholem vstupující do tohoto algoritmu bude námi zjištěný bod z podkapitoly 5.2.

S ohledem na vstupní data jsem se rozhodl použít pro nalezení nejkratších cest variantu Dijkstrova algoritmu s jedním zdrojovým uzlem. Tato varianta počítá nejkratší cesty z počátečního vrcholu do všech ostatních vrcholů v rámci grafu, přesně jak je potřeba. Navíc je zajištěno, že graf neobsahuje žádný cyklus záporné délky, protože jednotlivé váhy hran byly získány měřením vzdáleností mezi body. Záporná délka mezi dvěma body jistě změřena nebyla. Můžeme si tedy být jisti, že algoritmus bude pracovat rychleji, než ostatní dostupné algoritmy viz podkapitola 2.2.2.

Samotný algoritmus bude vyhodnocen pomocí knihovny BGL, která implementuje i další již zmíněné algoritmy. Tato knihovna je zaměřená na práci s grafy a grafovými algoritmy, proto je očekávána dobrá optimalizace a rychlost zpracování toho kroku.

## 5.6 Hledání cest s maximální délkou

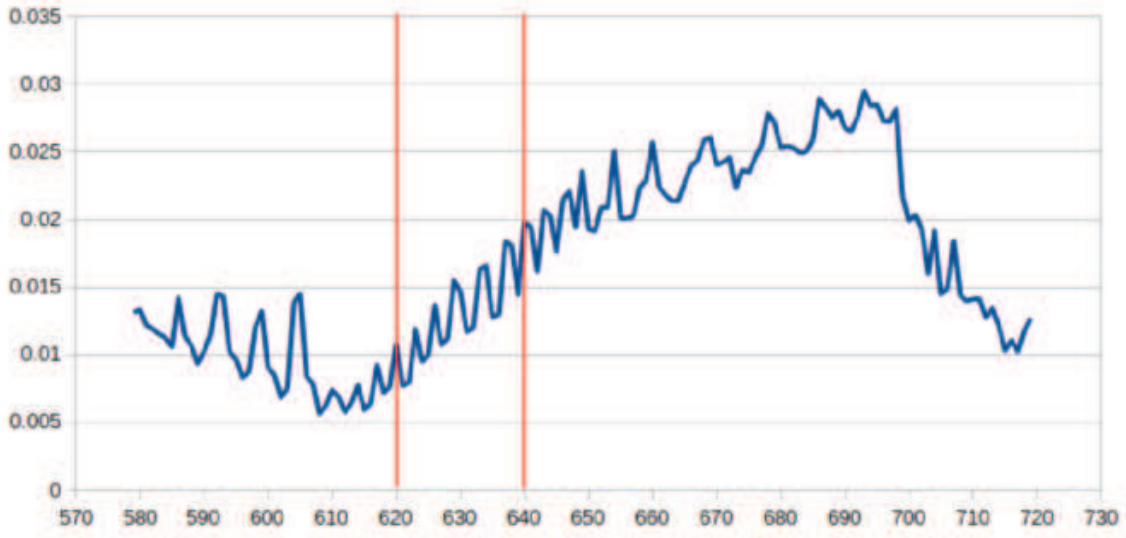
V této podkapitole bude navrženo jak najít cesty s nejdelší délkou. Hlavní motivací pro hledání těch nejdelších cest z centrálního bodu je fakt, že jednotlivé končetiny i hlava jsou od centrálního bodu vzdáleny více než jiné části těla. Předpokládá se, že vyneseme-li si jednotlivé délky cest do grafu na ose y a na osu x vyneseme index daného bodu, budou cesty končící v některé z končetin či oblasti hlavy spadat do některého z lokálních maxim takto vytvořené funkce.

Obrázek 10 vyobrazuje tento předpoklad. Stále je zde zásadní otázka: Jak vybrat ty správné lokální extrémy? V souboru přibližně 6000 bodů (tedy 6000 cest), není úplně triviální rozhodnout, co ještě lze považovat, nebo co ještě chceme považovat za lokální extrém.

Pro ukázku lze nahlédnout do obrazku 9. Lze vidět, že jednotlivé cesty se mohou lišit o velmi malou hodnotu, to by mohlo způsobit, že lokálních maxim bude velice mnoho. Toto lze částečně eliminovat nějakou předem vhodně zvolenou odchylkou.

Předpokládejme tedy, že pomocí této odchylky se podaří úspěšně odfiltrovat více než 80% falešných extrémů, díky studování a testování nad reálnými daty. I po tomto odfiltrování nám zbývá o něco méně než 20% extrémů, které již není tak snadné odfiltrovat. Je to dáno tím, že se při skenování Kinectem jednotlivé indexy ukládají po řádcích.

V našem případě jsme si vynesly na osu x indexy vrcholů a na osu y délky cest do těchto vrcholů. Existuje proto reálná možnost, že dva body mající dva po sobě jdoucí indexy, leží v mračnu jeden na okraji pravé ruky a druhý na okraji levé ruky (viz obr. 11/s30). To může opět způsobit nechtěný extrém ve funkci. Nelze proto jednoduše vzít všechny nejdelší cesty a pouze najít 5 cest s nejvyšší vzdáleností.




Obrázek 9: Ukázka oscilace hodnot v grafu.

Jak tento problém vyřešit? Napadly mě dva způsoby jak toto řešit.

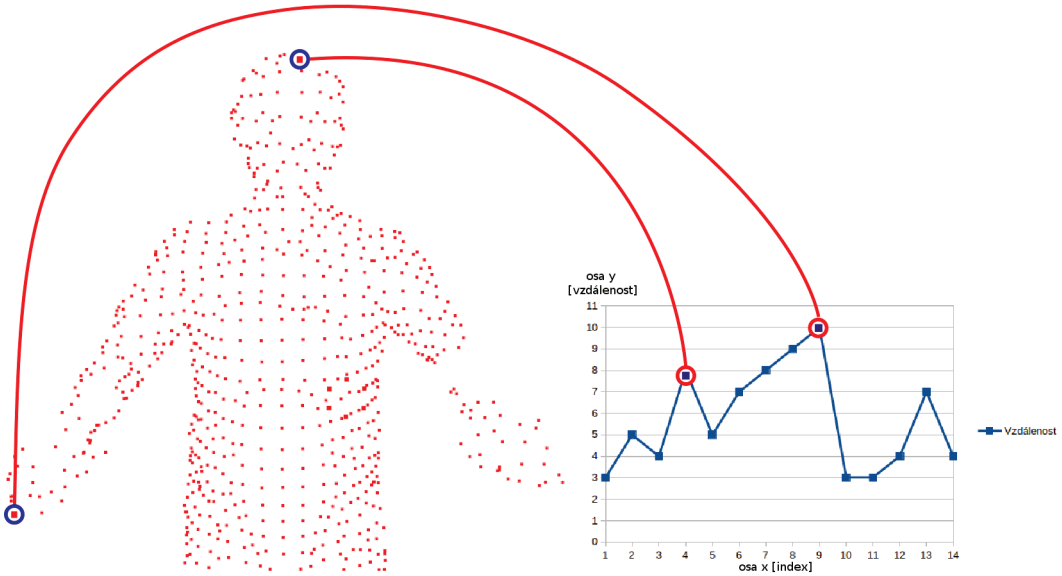
1. Eliminace špatných cest z odfiltrovaného souboru hodnot.

2. Vynést hodnoty délek cest, na osu x vzhledem k jejich vzájemné poloze.

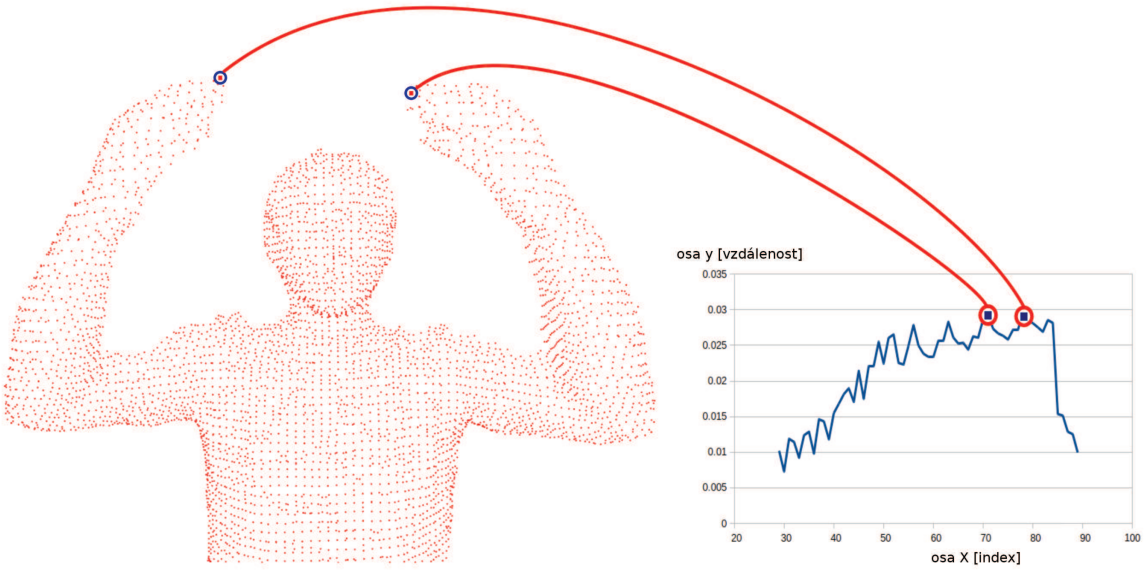
První způsob jsem implementoval a jeho návrh je popsán v následující kapitole 5.7. Druhý způsob bych zde rád nastínil. Hlavně z důvodu, že by se eventuálně dal použít při vylepšování této práce v budoucnosti. Moje představa o tomto řešení je, že by se jednotlivé body vynesli do grafu podle jejich polohy vzhledem k ostatním bodům v jejich okolí. Na osu X bychom tedy v grafu seřadili body podle toho jaké jsou indexy okolních bodů, toto okolí by muselo být vhodně zvolené.

Myslím si, že takto seřazené posloupnosti by v grafu utvořili jednodušeji rozeznatelné extrémy. Komplikací v tomto řešení je mapování bodů v mračnu. Protože budoucí algoritmy počítají s mapováním indexů od nuly do maximální hodnoty indexu. Přetřízením by se toto pořadí změnilo, bylo by proto nutné implementovat mapovací funkcí z nových indexů na původní a naopak.

Po předchozí podkapitole máme k dispozici *strom cest*. Víme tedy jak se z každého bodu dostat do bodu centrálního a navíc víme jak dlouhá daná cesta je. Využijeme-li dostupnou knihovnu *Persitence1D* (viz podkapitola 3.6) budeme mít velice efektivní implementaci hledání extrémů díky její asymtotické složitosti s jakou kód funguje a to i při takovém množství bodů, které se zpracovává. Rozhodl jsem se tuto knihovnu použít a následně extrahovat některé lokální extrémy, podle výše popsaného postupu.




Obrázek 10: Ukázka korespondence extrémů a bodů v mračnu.



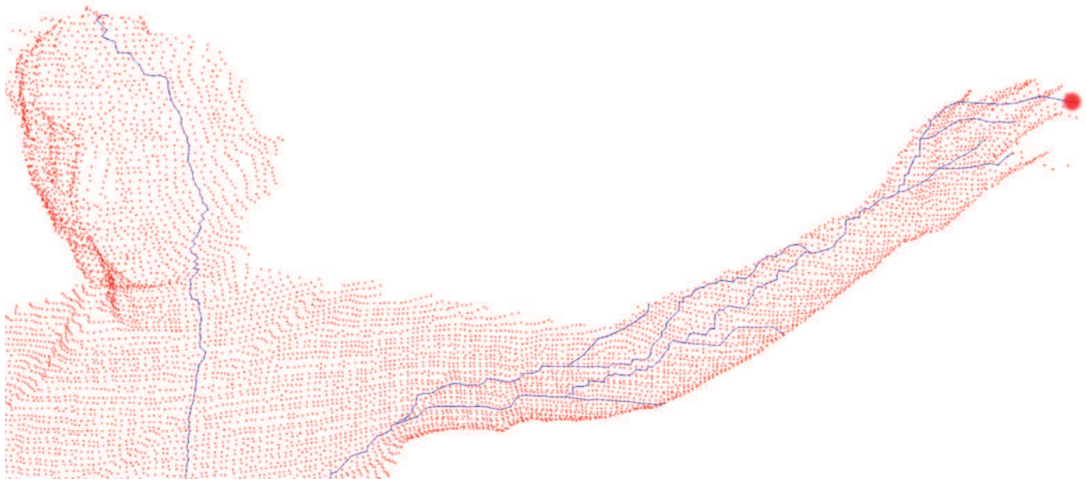

Obrázek 11: Ukázka možné příčiny problému v souboru hodnot.

## 5.7 Eliminace špatných cest

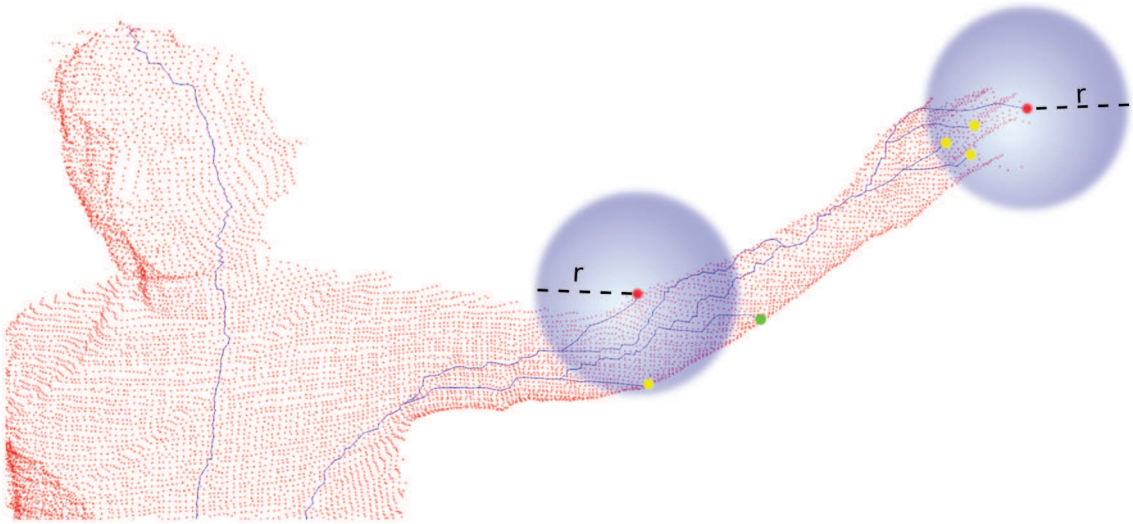
Hlavní důvod proč eliminovat cesty, byl zmíněn v předchozí kapitole. Zde bude popsána myšlenka jak filtrovat některé z cest, které byly v předchozím kroce vybrány jako kandidáti na cestu k nějaké končetině či hlavě. Dále budou uvedeny hlavní výhody a omezení tohoto postupu.

Na obrázku 12, lze vidět kus ruky a vyobrazené cesty, které byly vybrány jako extrémy. Jediná správná cesta označující tuto ruku je označena ve svém koncovém bodu červenou kuličkou. Potřebujeme eliminovat všechny ostatní cesty. Lze si všimnout, že v okolí konce správné cesty, končí i další tři jiné cesty, které však mají menší délku než cesta správná. Jednoduchým řešením by v takovém případě bylo, najít konce cest v okolí správné cesty a ty potom vyřadit ze seznamu kandidátních cest. Obecně je tedy možné eliminovat cesty tak, že se projdou všechny konce cest, prozkoumá se jejich okolí a cesty s menší vzdáleností se zahodí.

Definujme tedy dva koncové body zkoumaných cest $A$, $B$, kde $A \neq B$ a poloměr $r$. Postupně prohlašujeme koncové body kandidátních cest za bod $A$ a k němu ostatní body postupně za bod $B$. Zkoumáme-li potom body $A$ a $B$ a platí, že $|AB| < r$, potom bod $B$ vyjmeme ze seznamu kandidátních cest. Zde se opět využije struktura KD-tree a hledání sousedních bodů v mračnu. Tentokrát se nebude hledat $N$ sousedních bodů, ale budou se hledat všechny body v daném okolí o poloměru $r$. V seznamu výsledných bodů se mohou nacházet jakkékoliv body z mračna, které spadají do daného okolí a to i ty, které nejsou koncovým bodem, některé z kandidátních cest. Tyto body lze vyřadit z našeho návrhu na základě porovnání indexů jednotlivých bodů. Pokud tedy index zkoumaného bodu, nebude některý z indexů koncových bodů kandidátních cest, nebude se brát v úvahu (viz obr. 13/s31).



Obrázek 12: Ukázka možných cest a správné cesty.




Obrázek 13: Ukázka principu eliminace cest.

Tento postup nepokrývá úplně všechny možnosti. Je tak možné, že i přes eliminaci cest výše navrženým postupem, mohou zůstat v seznamu některé cesty, ikdyž by v seznamu být neměly (viz obr. 13/s31). Takových cest by obecně mělo být málo. Důležitým faktorem je zde poloměr $r$, ve kterém se hledají možné cesty k eliminaci. Tento poloměr nesmí být moc velký,

jelikož by docházelo k eliminaci cest i v jiných oblastech těla. Jako příklad uvádím, že pokud by poloměr byl větší než vzdálenost mezi chodidly, nebo obecně končetinami, mohly by se tímto způsobem eliminovat i správné cesty. Naopak, příliš malý poloměr by znamenal, velice nízkou míru zahazování špatně vybraných cest. Bude potřeba tento poloměr vhodně nastavit.

Problém zmíněný výše, který je také možné spatřit na obrázku 13, lze řešit. Pokud by podobných cest bylo opravdu jen malé množství, lze si představit, že se jednotlivé cesty projdou na základě známé fyziologické stavbě člověka. Například není možné nebo zcela běžné, aby hlava byla níže než dolní končetiny. Podobné zákonitosti by se dali do modelu zanést.

Dalším zajímavým řešením by mohlo být zjištění, zda některé cesty nesdílí část své trasy. To by mohlo například znamenat, že tyto cesty směřují do stejné končetiny, ale jedna znich končí dříve, než ta druhá. Takové cesty by se daly poté eliminovat tím, že by se ponechala pouze jedna z nich, například na základě jejich délky.

## 5.8 Označení částí těla

Základním cílem práce je označení oblasti hlavy a končetin těla v hloubkové mapě. Cílem této podkapitoly bude navrhnout, jak z dosud analyzovaných dat dosáhnout tohoto základního cíle. K dispozici máme seznam koncových bodů jednotlivých cest a také známe jednotlivé předchůdce těchto bodů. Ideálně máme pět cest, které jednotlivě vedou do levé dolní končetiny, pravé dolní končetiny, levé horní končetiny, pravé horní končetiny a oblasti hlavy.

K označení jednotlivých koncových bodů můžeme použít opět knihovnu PCL. Ke každému koncovému bodu vložíme do hloubkové mapy malou kouli o rozměrech několika málo centimetrů. Z koncových bodů také vykreslíme jednotlivé úseky cest do centrálního bodu.

# 6 Implementace

V následujících odstavcích bych se chtěl zmínit o použitých vstupních datech, nejdůležitšjších funkcích a nastaveních různých metod. K implementaci navržených částí jsem si vybral programovací jazyk C++. Zásadním důvodem k použití tohoto jazyka byla dostupnost knihoven pro práci s Kinectem na operačním systému Linux. Dále pak fakt, že běh programů psaný v C++je rychlý a pro výpočetně náročné aplikace a realtime aplikace je toto kriérum velmi důležité. Při implementaci jsem použil standard z roku 2011.

V rámci překladu je použit software CMake. Dle [8, 9] se jedná o cross-platform software, pro automatizaci překladu. CMake nám umožňuje generovat výstup do určené složky a výsledkem mohou být projekty pro různé vývojové studia, jako Microsoft Visual Studio, Eclipse, Code Blocks, nebo také soubor Makefile. Hlavním konfiguračním souborem je soubor CMakeLists.txt. V něm lze například definovat požadované minimální verze různých knihoven, samotného CMake nebo také podmíněný překlad.

## 6.1 Vstupní data a jejich formát

Vstupní data jsou nahrána pomocí aplikace `libfreenect2pclgrabber`. Ta pořizuje snímky pomocí Kinectu a zapisuje je do adresáře zvoleného pomocí přepínače `-f`. Knihovna PCL používá formát PCD (Point Cloud Data). S vývojem knihovny PCL se vyvíjel i tento formát, a proto je možné se setkat s různými verzemi. Jak data vypadají v prohlížeči PCD souborů nám ukazují obrázky 3 a 12. Každý soubor obsahuje hlavičku, která udává:

1. Verzi PCD souboru
2. Jakým způsobem jsou reprezentovány body
   (a) Souřadnice $[x, y, z]$
   (b) Souřadnice $[x, y, z]$ a barva bodu
   (c) Souřadnice $[x, y, z]$ a povrchové normály
   (d) A další...
3. Velikost dimenze v bytech
4. Datový typ
5. Šířka mračna
6. Výška mračna
7. Pohled na mračno
8. Počet bodů
9. Způsob uložení PCD (binární nebo ASCII)

## 6.2 Použité datové struktury

V programu je použito několik struktur, jejichž názvy budou použity v dalších podkapitolách. Z tohoto důvodu zde jednotlivé struktury popíši a vysvětlím jejich funkci.

1. class pcl :: PointXYZRGB
2. class pcl :: PointCloud
3. class pcl :: VoxelGrid
4. class SimpleOpenNIViewer
5. struct ___edge_data
6. boost :: adjacency_list Graph

### 6.2.1 PointXYZRGB

Struktura *pcl::PointXYZRGB* reprezentuje Euklidovské souřadnice xyz a RGB informaci o barvě jednoho bodu. Barva je zabalena do typu int a následně přetypována na typ float z historického hlediska viz dokumentace [11].

### 6.2.2 PointCloud

Dokumentace [13] říká, že třída *pcl::PointCloud* reprezentuje základní třídu v PCL pro ukládání množiny bodů ve 3D. Je implementována jako šablona, což umožňuje specifikovat typ použitého typu bodů. Každý PointCloud*(dále jen mračno)* má tyto vlastnosti:

1. width - specifikuje šířku mračna počtem bodů
   (a) pro neuspořádaná mračna tento parametr znamená celkový počet bodů v mračnu
   (b) pro uspořádaná mračna je význam tohoto parametru *počet bodů v jednom řádku*
2. height - specifikuje výšku mračna počtem bodů
   (a) pro neuspořádaná mračna je tento parametr nastaven na 1
   (b) pro uspořádaná mračna je význam tohoto parametru *celkový počet řádků* v mračnu
3. points - jedná se o datové pole obsahující všechny body mračna jednoho typu, obecně PointT
4. is_dense - tento parametr říká zda všechny body obsahují konečnou hodnotu (tehdy je nastaven na true) a nebo ne například Inf a NaN hodnoty (poté je nastaven na false)
5. sensor_origin_ - specifikuje počátek senzoru při sběru dat
6. sensor_orientation_ - specifikuje rotaci senzoru při sběru dat

Dle dokumentace [12] třída *pcl::VoxelGrid* sestrojí voxelovou mříž nad daným mračnem. Poté v každém voxelu podvzorkovává data (body) a nahradí je jejich těžištěm. Tento postup je pomalejší než nahrazování středem každého voxelu, ale o to přesněji reprezentuje daný povrch.

Třída *SimpleOpenNIViewer* [13] slouží v programu k proudovému zpracování mračen. Obsahuje metodu void run(), ve které se nejdříve vytváří rozhraní OpenNIGrabber. Toto rozhraní umožňuje přijímat proud dat z kamer, které jsou kompatibilní s OpenNI (Kinect, Asus Xtion Pro a další). Dále je v metodě vytvořen *callback* na funkci void cloud_cb_ (const pcl :: PointCloud <pcl::PointXYZRGB>::ConstPtr &cloud), kde se předává i reference na OpenNIViewer třídu. Ten je zaregistrován ve třídě PCDGrabber. Třída má implementovánu i metodu void set_stream(vector< string> pcd_files) na přiřazení souborů PCD.

Struktura *___edge_data* má jedinou položku a to double rate. Jedná se o délku hrany, kterou daná struktra charakterizuje. Tato struktura je nutná pro použití s knihovnou BGL, při použití vlastních datových typů, jiných než základních v BGL.

```
struct __edge_data
{
    double rate;
};
```

Výpis 1: Struktura hrany grafu pro BGL

Seznam sousedů *boost::adjacency_list Graph* je struktura definovaná pro použití s knihovnou BGL. Její definice je: typedef boost :: adjacency_list <boost::vecS, boost :: vecS, \boost :: undirectedS , boost :: no_property,___edge_data> Graph;
*Adjacency_list* je třída reprezentující sekvenci výstupních hran pro každý vrchol v grafu. Definice této třídy je adjacency list <EdgeList, VertexList , Directed , VertexProperties , EdgeProperties , GraphProperties>, kde *EdgeList* řídí jaký typ kontejneru je použit pro uchování sekvence výstupních hran pro každý vrchol. *VertexList* řídí jaký typ kontejneru je použit pro uchování sekvence vrcholů. *Directed* paramter umožňuje definovat zda je graf orientovaný, neorientovaný nebo obousměrný (v mém případě používám neorientovaný graf). *VertexProperties* specifikuje vnitřní reprezentaci vrcholu. *EdgeProperties* specifikuje vnitřní reprezentaci hrany (v mém případě struktura ___edge_data) a *GraphProperties* specifikuje vnitřní reprezentaci grafu (v mém případě nenastaveno).

## 6.3 Základní nastavení

V programu je možné nastavit všechny hlavní části, které mají nějaký vliv na přesnost či rychlost výpočtu. Převážně bylo toto nastavení implementováno ke snadnějšímu testování a hledání optimálního nastavení jednotlivých částí programu, bez nutnosti překládat zdrojový kód. Kapitola 7 věnující se experintům bude později rozebírat jednotlivá nastavení s ohledem na vstupní data, rychlost a požadovanou přesnost výsledku.

```
string pcd_files = "test_data"; // název složky s daty
float limit = -1.0;      // limit
float leaf_size = 0.015f; // velikost jednoho voxelu v~metrech
float persistence = 0.01;
float radius = 0.1;      // poloměr v~jakém se hledají cesty k~eliminaci
int num_neighbors = 0;   // počet hledaných nejbližších bodů
int source_index = 0;    // index zdrojového vrcholu
int target_index = 0;    // index cílového vrcholu
bool display = false;    // indikátor zda chceme spustit vizualizaci výsledku
bool verbose = false;    // indikátor vypisování ladících výpisů
bool experiments = false; // indikátor režimu pro experimenty
```

Výpis 2: Základní nastavení programu

Tyto proměnné se nastavují v závislosti na předaných parametrech programu model_driver viz příloha B. K vyhodnocení předaných parametrů jsem použil knihovnu optionparser.h [15]. Pomocí této knihovny lze jednoduše definovat jednotlivé parametry, zda jsou povinné nebo nepovinné nebo jestli je očekávána nějaká jeho hodnota. Jednoduše tak definujeme požadované parametry, jejich nastavení a knihovna nám je již zpracuje. V případě chyby vypíše nápovědu programu. Jednotlivé hodnoty parametrů se musí načíst ručně do připravených proměnných. Výše uvedené hodnoty jsou nastaveny již od počátku programu a není nutné je uvádět při spuštění programu. V následující ukázce je příklad načítání numerické hodnoty parametru pomocí funkce strtof ze standardní knihovny jazyka C++.

```
case LEAF:
   fprintf(stdout, "--leaf with argument '%s'\n", opt.arg);
   leaf_size = std::strtof(opt.arg, &end);
   if(leaf_size == 0 || leaf_size == HUGE_VAL || leaf_size == HUGE_VALF ||
       leaf_size == HUGE_VALL){
      std::cout << "range error, got ";
      return EXIT_FAILURE;}
   break;
```

Výpis 3: Příklad zpracování argumentů programu

## 6.4 Základní funkce programu

Podle zadaných parametrů při spuštění se spustí jedna z následujících funkcí programu:

1. void show_center_point(const char *file_name, float limit , bool display )
2. void render_cloud(const char *file_name, float limit , float leaf_size )
3. void compute_skeleton(const char *file_name, float limit , float leaf_size , float persistence , float radius , bool display , bool verbose , int num_neighbors)
4. SimpleOpenNIViewer.run()

Funkce show_center_point slouží zejména k testovacím účelům. Tato funkce vypočítá index centrálního bodu v zadané hloubkové mapě. Index vypočteného bodu je vypsán na standardní výstup. V případě, že je zadán této metodě parametr display s hodnotou true, vytvoří se vizualizační prostředí z knihovny PCL, které je reprezentováno třídou pcl :: visualization :: PCLVisualizer. Do instance této třídy se přiřadí námi používané mračno, nastaví se pozice a směr kamery. Následně se na souřadnice bodu s výsledným indexem přidá červená koule o poloměru $r$, kde $r = 0.5 * leaf\_size$. Mračno si lze prohlížet ve vytvořeném visualizéru. Ovládání je popsáno v příloze C.

Druhá ze zmíněných funkcí byla vytvořena také k testovacím účelům a slouží k prohlížení jednotlivých mračen. Tato funkce vyžaduje předání jména mračna, velikosti voxelu pro KD-tree a vzdálenosti, od které se do výpočtu nezapočítávají jednotlivé body. Funkce poté spustí visualisér, stejně jako v případě funkce show_center_point, ale je použito základní nastavení třídy pcl :: visualization :: PCLVisualizer.

Nejdůležitější z uvedených funkcí je compute_skeleton. Ta má za úkol vypočítat polohy jednotlivých končetin, oblast hlavy a také centrální bod těla. Hlavním parametrem této funknce je const char *file_name, což je název souboru obsahující hloubkovou mapu ve formátu PCD. Dále je nutné předat funkci float limit pro vyjmutí bodů za touto hranicí, float leaf_size pro definování velikosti jednoho voxelu, float persistence sloužící k nastavení odchylky, float radius je poloměr pro hledání nejbližších bodů, bool display zapíná nebo vypíná vizualizaci výsledku bool verbose obsluhuje výpis ladících výpisů na standardní výstup a int num_neighbor nastavuje počet hledaných nejbližších bodů.

Poslední možností, kterou program nabízí je spuštění metody run třídy SimpleOpenNIViewer. Této třídě se předá seznam jednotlivých PCD souborů a nad nimi je poté spuštěna jedna z výše uvedených funkcí. Jedná se o náhradu real-time zpracování snímků přímo z Kinectu. Jedním důvodem bylo testování aplikace bez nutnosti mít připojený Kinect během běhu programu a druhým důvodem byla rychlost zpracování jednotlivých metod. Rychlost metod závisí na předaných parametrech.

## 6.5 Výpočet kostry

Tato podkapitola se detailně věnuje funkci compute_skeleton a všem implementovaným a použitým algoritmům v této funkci. Vstupy této funkce byly popsány v předchozí části, a proto lze přejít k popisu funkce samotné.

### 6.5.1 Načtení vstupů

Na začátku funkce viz výpis 4 jsou vytvořeny dva objekty typu pcl :: PointCloud. Do prvního z těchto objektů se načte hloubková mapa pomocí funkce pcl :: io :: loadPCDFile. Tato metoda vyžaduje jako první parametr cestu k souboru a jako druhý parametr objekt typu PointCloud.

```
pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud (new pcl::PointCloud<pcl::
    PointXYZRGB>);
pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud_tmp (new pcl::PointCloud<pcl::
    PointXYZRGB>);
pcl::io::loadPCDFile (file.c_str(), *cloud);
```

Výpis 4: Ukázka načtení mračna skrz PCL

### 6.5.2 Úprava mračna

Načtené mračno dále předáme do funkce cut_points jako první parametr viz výpis 9. Druhým parametrem je objekt typu PointCloud do kterého se uloží výsledek. Třetí parametr této funkce je limit , který definuje hranici pro zahazoní bodů. Ze vstupního mračna se tedy odfiltrují všechny body, které jsou dále než hodnota daná parametrem limit ve směru osy Z. Následně smažeme obsah původního mračna pomocí metody clear (), aby se nepromíchala data v následující funkci sifter (). Stejně jako do předchozí funkce předáme této funkci vstupní a výstupní mračno bodů a jako třetí parametr je předána proměnná leaf_size . Ta definuje velikost voxelu ve struktuře pcl :: VoxelGrid. Princip fungování je popsán v kapitole 5.2.

```
cut_points(cloud, cloud_tmp, limit);
cloud->clear();
sifter(cloud_tmp, cloud, leaf_size);
```

Výpis 5: Ukázka úpravy mračna

### 6.5.3 Vytvoření grafu z mračna

Základní operace nad mračnem jsou v této chvíli již hotovy a dále se bude mračno již používat pouze jako zdroj informací bez úprav jeho samotného. Následuje výpočet centrálního bodu pomocí funkce compute_center(). Základní myšlenka je popsána v podkapitole 5.2.

Další fází je vytvoření grafu pro knihovnu BGL z našeho mračna viz výpis 6. Návrh této části je v podkapitole 5.4. Graf je typu Graph viz podkapitola 6.2 a jeho velikost je stejná jako počet bodů v mračnu. Jsou definovány dva vektory. První vektor neighbors obsahuje indexy bodů nalezených nejbližších bodů, jeho velikost je o jedna větší než definovaný počet hledaných nejbližších bodů. Je tomu tak proto, že funkce nearestKSearch vrací jako jeden z nalezených $K$ nejbližších bodů zdrojový bod. Druhý vektor udržuje vzdálenosti těchto $K + 1$ bodů od zdrojového bodu. Definovaná je i pomocná hrana tmp_edge k pozdějšímu vložení do grafu.

Poté se prochází jednotlivé body v cyklu for (řádky 5 až 16) a pro každý bod $i$ se hledá $K+1$ nejbližších bodů. Opět se hledá o jeden bod více, protože prvním nalezeným bodem je vždy bod zdrojový. Pokud je počet nalezených bodů kladný, prochází se potom vektor nalezených bodů cyklem for (řádky 9 až 12). Pro každý takový bod $j$ se potom sestrojí hrana od zdrojového bodu $i$ k právě vybranému bodu $j$ a přidá se do grafu $g$ s ohodnocením neighbors[j], což je vzdálenost bodu $j$ od bodu $i$. Navíc ještě dochází k testu zda stejnou hranu již graf obsahuje. Pokud je tento test pozitivní hrana se do grafu podruhé nepřidá.

```
Graph g(cloud->size());
std::vector<int> neighbors (num_neighbors + 1);
std::vector<float> distances_tmp (num_neighbors + 1);
__edge_data tmp_edge;
for(size_t i = 0; i < cloud->size (); ++i)
{
    if (kdtree.nearestKSearch(*cloud,i, num_neighbors + 1, neighbors,
        distances_tmp) > 0)
    {
        for(size_t j = 1; j < neighbors.size(); ++j){
            if(!boost::edge(i, neighbors[j], g).second){
                tmp_edge.rate = distances_tmp.at(j);
                boost::add_edge(i, neighbors[j], tmp_edge, g);
            }
        }
    }
}
```

Výpis 6: Sestrojení grafu pro výpočet nejkratších cest

### 6.5.4 Shortest–Path First

Nad vytvořeným grafem lze již spustit funkci z knihovny BGL boost:: dijkstra_shortest_paths . Dle [2] lze tuto metodu spustit dvěmi způsoby. První způsob je specifický tím, že se předává parametr distance_map(). Tímto zajistíme, že se nejkratší cesta ze zdrojového vrcholu do všech ostatních vrcholů zapisovat do *distance map (mapy vzdáleností)*. Druhou možností pak je zaznamenávat tyto cesty v mapě předchůdců. Pro každý vrchol $u \in V$, je $\pi[u]$ předchůdcem $u$ ve stromu nejkratších cest (až na vyjímku $\pi[u] = u$, kdy je $u$ samotným zdrojovým vrcholem nebo se jedná o vrchol, který není dostupný ze zdrojového vrcholu). Dále si lze vytvořit vlastní metodu pro zaznamenávání předchůdců. Já jsem se rozhodl použít PredecessorMap podle dokumentace knihovny [10]. Metoda z této knihovny funguje podle následujícího pseudo-kódu (převzato z [2]), kde $w$ značí váhu hrany, $d$ je vzdálenost a $\pi$ je předchůdce každého uzlu. $Q$ je prioritní fronta umožňující operaci DECREASE-KEY:

---

```
DIJKSTRA(G,s,w)
for each vertex u ∈ V
    d[u] ← ∞
    π[u] ← u
d[s] ← 0
INSERT(Q, s)
while(Q ≠ ∅)
    u ← EXTRACT-MIN(Q)
    S ← S ∪ u
    for each v ∈ Adj[u]
        if(w(u, v) + d[u] < d[v])
            d[v] ← w(u, v) + d[u]
            π[v] ← u
            if(color[v] = WHITE)
                color[v] ← GRAY
                INSERT(Q, v)
            else if(color[v] = GRAY)
                DECREASE-KEY(Q, v, w(u, v) + d[u])
        else
            ...
    color[u] ← BLACK
return(d, π)
```

---

Dle uvedeného pseudo-kódu jsou všechny nejkratší cesty ze zdrojového vrcholu do všech ostatních vrcholů nalezeny postupným zvětšováním množiny vrcholů $S$ do kterých algoritmus zná nejkratší cestu. V každém kroku je další vrchol přidán do $S$ dle prioritní fronty.

Tato fronta obsahuje vrcholy $V$, které jsou prioritizovány podle vzdálenosti, což je délka nejkratší cesty, která byla doposud zjištěna pro všechny vrcholy. Poté je vrchol $u$ přidán z vrcholu prioritní fronty do $S$. Každá hrana vycházející z uzlu $u$ se prochází a pokud je součet vzdálenosti z vrcholu $u$ a hrany $(u, v)$ menší než současná vzdálenost ve vrcholu $v$, tak je vzdálenost pro vrchol $v$ zmenšena. Poté se algoritmus vrací zpátky k vrcholu prioritní fronty a pokračuje dalším vrcholem, který je na vrcholu fronty. Algoritmus pokračuje do té doby, než je prioritní fronta prázdná. Tento algoritmus používá tři barvy k označení, do jaké množiny určitý uzel patří. Vrcholy označeny černou barvou jsou v množině $S$. Vrcholy označené šedou nebo bílou barvou jsou ve $V$. Bílou barvou jsou označeny ty vrcholy, které ještě nebyly ještě objeveny a šedou ty, které jsou v prioritní frontě.

### 6.5.5 Hledání extrémů

K nalezení jednotlivých končetin a hlavy v hloubkové mapě se v programu využívá hledání extrémů (návrh viz podkapitola 5.6). Extrémy jsou hledány v cestách, které vrátila funknce boost :: dijkstra_shortest_paths . K nalezení extrémů jsou v rámci programu implementovány dvě funkce void find_maximum(float limit) a void find_extremes( vector<double> *in_values, unsigned int size , float persistence , bool verbose , vector<int> *out_maxs). V rámci experimentů v kapitole 7 bude vyhodnoceno, která varianta vrací lepší výsledky. Oboum variantám se předává vector<double> *in_values, ve kterém daná funkce má najít lokální extrémy (maxima). Tento vektor obsahuje vzdálenosti jednotlivých vrcholů v grafu pro Dijkstrův algoritmus nejkratších cest. Obě funkce mají za úkol vrátit pole indexů nalezených extrémů v parametru vector<int> *out_maxs.

První varianta je založena na porovnávání dvou po sobe jdoucích hodnot. Protože je nutné z ohlednit možnou oscilaci mezi sousedními hodnotami, je porovnání dvou hodnot zjemněno pomocí odchylky viz výpis 7. Pokud tedy jsou porovnávané hodnoty od sebe vzdáleny méně a nebo stejně jako definovaná odchylka epsilon , je prohlášeno že se navzájem rovnají.

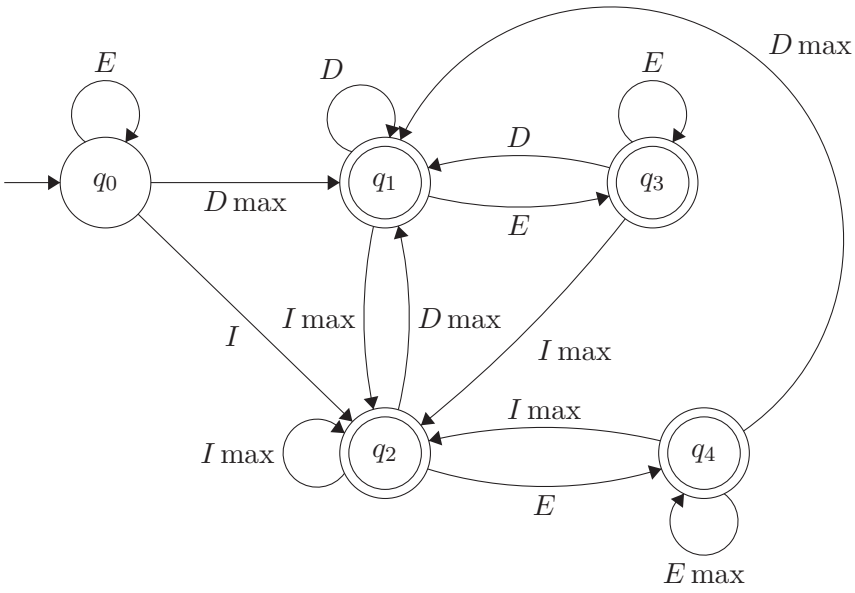
```
bool almost_equal(double A, double B, double epsilon){
   if (fabs(A - B) <= epsilon)
      return true;
   else
      return false;
}
```

Výpis 7: Porovnání dvou hodnot typu float

Jádro funkce je pak založeno na stavovém automatu viz obrázek 14. Hrany jsou označeny písmeny $E(equal), D(decrease), I$ $(increase)$ a jejich význam reflektuje zda právě porovnávané hodnoty jsou vyhodnoceny stejné $(E)$ nebo je aktuální hodnota menší než předchozí $(I)$ nebo je aktuální hodnota větší než předchozí $(D)$. Příjmající jsou všechny stavy kromě stavu prvního, protože se předpokládá, že porovnáváme vždy minimálně desítky hodnot. Tento předpoklad je podložen faktem, že naše mračno má alespoň stovku bodů (standardně několik tisíc) a v takovém

souboru jsme počítali nejkratší cesty z jednoho bodů do všech ostatních. Má-li tedy mračno $N$ bodů, potom nalezených cest z jednoho bodu, do všech ostatních bodů je $N - 1$. Možnost, že by hodnoty byly natolik vyrovnané, aby všechny body byly vyhodnoceny jako stejné záleží na zvolené odchylce. Tato odchylka je nastavena podle nejmenší a největší hodnoty v souboru poměrnou částí jejich rozdílu. Obecně je v tomto algoritmu extrém hlášen tehdy, pokud předchozí porovnání bylo *increase* a aktuální je *decrease*. Mezi těmito porovnáními je dále povoleno libovolné množství porovnání *equal*. Dále je vyhodnocen extrém tehdy, pokud počáteční hodnoty jsou vyhodnocovány jako *equal* a aktuální porovnání hodnot jako *decrease*. Podobně je ošetřen konec souboru hodnot, kdy je extrém uložen tehdy, pokud hodnoty byly vyhodnocovány jako *increase* až do konce souboru, nebo pokud konec souboru byl vyhodnocen jako *equal* ale tomu předcházelo pouze vyhodnocení jako *increase*. Případy, kdy je v algoritmu hlášeno maximum, jsou u hran označeny popiskem *max*.

Stavy automatu mají také svůj vlastní význam. Kromě vstupního stavu, je zde stav $q_1$, do kterého se přechází vždy, když je porovnání vyhodnoceno jak *decrease*. Stav $q_2$ má podobnou roli jako stav $q_1$ ale přechází se do něj, když výsledek porovnání je *increase*. Dále má automat dva stavy $q_3$ a $q_4$, které udržují informaci, že porovnané hodnoty jsou stejné. Tyto dva stavy se liší tím, zda jeho předchůdce byl stav $q_1$ nebo $q_2$.




Obrázek 14: Stavový automat hledání extrémů

Druhá varianta hledání extrémů využívá externí knihovnu `persistence1d.hpp`. Tato knihovna umí najít jak lokální minima, tak maxima v souboru dat. Lze si definovat hodnotu, kterou knihovna nazývá `persistence`. Tato hodnota je určitá odchylka mezi dvojicí minima a maxima.

### 6.5.6 Eliminace extrémů

Eliminaci extrémů jsem popsal v kapitole 5.7, zejména z hlediska proč se tato eliminace dělá. Z důvodu zjednodušení zde neuvádím přesný zdrojový kód, ale jen psedo-kód viz výpis 8. Na začátku máme k dispozici seznam extrémů max_indexes. Dále nastavuji pomocné proměnné max a max_index na nízké hodnoty, protože do proměnné max bude uložena maximální délka v každém kroku algoritmu a do proměnné max_index bude uložen index bodu, který má právě maximální délku. Dále je nastavena ukončovací podmínka cyklu end_cond.

V každé iteraci se postupně vybírají body z listu max_indexes a pro každý takový bod se hledají body ve vzdálenosti danou proměnnou radius. Tyto body se poté procházejí a pokud je některý z bodů v listu max_indexes, porovná se jeho vzdálenost se současnou maximální vzdáleností. Pokud je větší, uloží se tato vzdálenost jako aktuálně největší vzdálenost do proměnné max a jeho index do proměnné max_index. Pokud je menší, znamená to, že jej chceme vyřadit, a proto jej vložíme do pomocného listu points_to_remove a nastavíme ukončovací podmínku cyklu.

```
max_indexes = list(EXTREMES)
bool end_cond = False
points_to remove = list()
while not end_cond:
   end_cond = True
   max = -FLT_MAX
   max_index = -1
   foreach P in max_indexes:
      max = -FLT_MAX
      max_index = -1;
      foreach SP in kdtree_search_in_radius:
         if is SP in max_indexes:
            if SP.get_distance() > max:
               max = SP.get_distance()
               max_index = SP.get_index()
            else:
               end_cond = False
               points_to_remove.pushback(SP)
      if not end_cond:
         break;
  foreach P in points_to_remove:
      if is P in max_indexes:
         if P.get_index() != max_index:
            max_indexes.remove(P.index)
```

Výpis 8: Pseudo-kód algoritmu eliminace extrémů

Potom co se projdou všechny body z výstupu KDtreeSearch zkontrolujeme podmínku, zda se nějaký bod přidal do seznamu bodů k vyřazení. Pokud ano ukončuje se vnitřní cyklus a body v tomto poli se poté smažou z listu max_indexes. Jinak se pokračuje dále dokud se neprojdou všechny body.

Výše uvedený pseudo-kód neobsahuje řešení toho, že se odebírají položky z listu, nad kterým se právě iteruje. Jedná se o list max_indexes. Toto jsem vyřešil změnou indexů ve chvíli, kdy se odebírá daný bod z tohoto listu. V rámci algoritmu se musí zmenšit index vnějšího for cyklu a také vnitřního.

### 6.5.7 Sestrojení cest

V této podkapitole se budu věnovat samotnému sestrojení cest.

```
vector<PathType> paths_to_max;
for(vector< int >::iterator it = max_indexes.begin(); it != max_indexes.end();
    it++){
   PathType path;
   Vertex v~= getMyVertex(g, (*it));
   for(Vertex u~= p[v]; v~!= s; v~= u, u~= p[v]) {
      std::pair<Graph::edge_descriptor, bool> edgePair = boost::edge(u, v, g);
      Graph::edge_descriptor edge = edgePair.first;
      path.push_back( edge );
    }
    paths_to_max.push_back(path);
}
```

Výpis 9: Sestrojení cest ze struktury Graph

Vstupem do této částí je seznam extrémů. Tento seznam obsahuje indexy těchto extrémů. Každý extrém je konec nějaké cesty v sestrojeném grafu. Pro každý tento extrém se musí sestrojit cesta od tohoto extrému ke zdrojovému uzlu. Každá cesta PathType path je sestrojena tak, že se nastaví aktuální vrchol Vertex v na cílový uzel. A poté se postupuje směrem ke zdrojovému uzlu přes předka daného uzlu a postupně se přidávají uzly do cesty path.

# 7 Experimenty

V této části budou popsány experimenty, použitá testovací data a natáčená scéna, metriky pro pozdější vyhodnocení, použitá nastavení programu a průběh experimentů. Experimenty probíhaly na následující sestavě se 4 jádrovým procesorem Intel Core i7-3537U CPU s frekvencí jádra 2.00GHz, grafickou kartou NVidia GF117M a 8GB RAM s použitím Microsoft Kinect druhé generace.

## 7.1 Návrh experimentů

V následujících odstavcích jsou přesně definovány jednotlivé parametry použité pro běh experimentů, popsány a definovány použité metriky pro vyhodnocení a porovnání výsledků. Je popsáno jaká testovací data byla při experimentech použita, a jak vypadala nahrávací scéna. Testovací data jsou obsahem přílohy DVD.

### 7.1.1 Testovací data a scéna

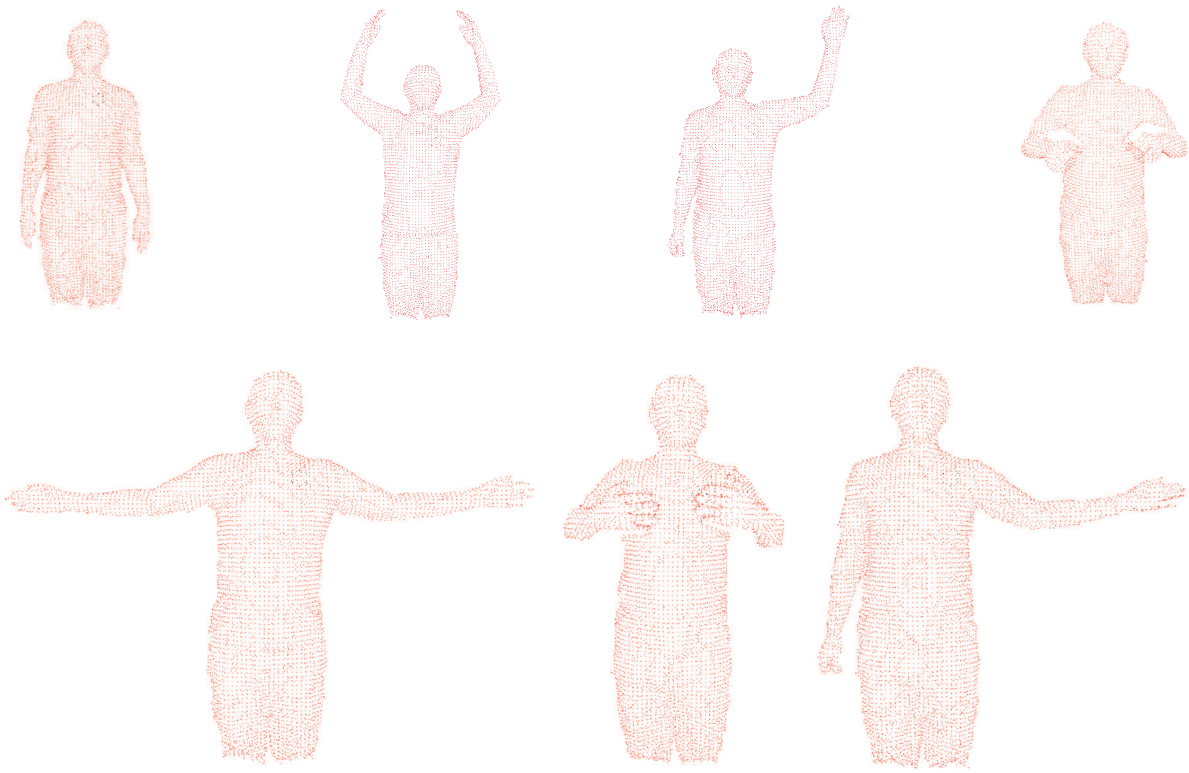
Použita budou k experimentům reálné snímky nahrané senzorem Kinect druhé generace. Budou použita jak k otestování návrhu mé metody, tak k měření a porovnání s existujícími metodami.

Celkově je v testovací množině 1211 snímků s jedinou osobou stojící před rovnou plochou. Barevná složka jednotlivých bodů je nevýznamná, a proto se zobrazují všechny body červeně na bílém pozadí. Jedním z důvodů je i tisk této práce, ale v rámci nastavení lze zvolit i barevný mód, který zachovává barvy jednotlivých bodů.

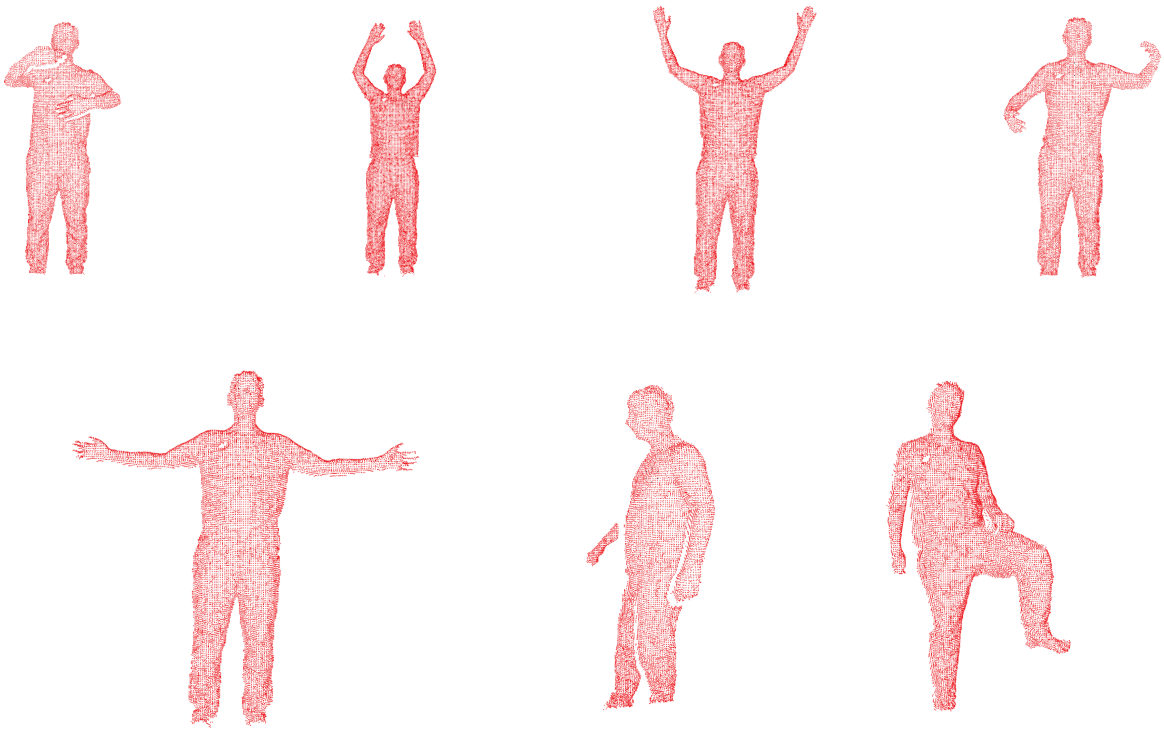
Testovací data jsou rozdělena na dvě části. První část zahrnuje 448 snímků. Postava na snímcích stojí čelem ke kameře a nerotuje kolem své osy. Tato testovací sada obsahuje nasnímanou postavu od hlavy až po kolena. To by mělo poskytovat lepší představu o tom, jak se program bude chovat při vyhodnocování osob, kterým nejsou zcela vidět nohy. Během času zaujímá postava několik základních póz:

1. Ruce podél těla.
2. Ruce nad hlavou
3. S rozpaženýma rukama.
4. S jednou rukou upaženou a druhou rozpaženou.
5. S jednou rukou upaženou a druhou nad hlavou.
6. Ruce zaujímají polohu držení volantu.
7. Ruce před hrudníkem.

Obrázek 15: Ukázka reálných dat bez detekce a s detekcí



Obrázek 16: Ukázka testovacích dat ze sady č.1



Obrázek 17: Ukázka testovacích dat ze sady č.2

Druhá testovací množina obsahuje 763 snímků. Stejně jako v množině první je na snímcích pouze jediná osoba stojící před rovnou plochou. Osoba zaujímá bohatší spektrum póz a rotuje přibližně o -+45 stupňů do stran. Na snímcích dochází i k překryvům obou ruk osoby. Tentokráte je osoba nasnímaná od hlavy až po chodidla.

### 7.1.2 Metriky

Jednotlivé metriky budou sloužit k vyhodnocení funkčnosti programu, rychlosti zpracování jednoho snímku a k porovnání s existujícími řešeními. Hlavní metrikou bude průměrný počet označených částí na snímcích. Měření průměrného rozdílu vzdáleností jednotlivých částí těla a označení těchto částí programem, není triviální, jelikož jsou data reálná a nejsou syntetická. Nejsou tedy známy přesné body, které by měly být označeny. Z tohoto důvodu budou ručně vybrány body na desítce reprezentativních snímků, a poté pomocí programu změřeny vzdálenosti mezi označenými body programem a ručně vybranými body. Tyto rozdíly vzdáleností budou vyhodnocovány pro jednotlivé oblasti zvlášť, a také bude vyhodnocena i celková přesnost ve všech oblastech. V rámci vyhodnocení budou označená slabá a silná místa.

1. Rozdíl vzdáleností mezi označenými body programem a ručně vybranými body. Tato metrika bude do jisté míry ilustrativní, jelikož ruční výběr může zahrnovat určitou chybu, která může být způsobena jednak výběrem snímků, ale také i samotným označením správného bodu v mračnu, vzhledem ke kterému se má počítat odchylka.

2. Čas zpracování jedno snímku v závislosti na přesnosti a použitých nastaveních. Za tento čas se bude považovat čas strávený procesorem nad zpracováním jednoho snímku.

3. Průměrný počet označených částí těla v dané testovací množině. Za dokonalý výsledek lze považovat označení pěti částí těla (hlava, dvě horní a dvě dolní končetiny) a centrálního bodu. Celkem tedy šesti částí těla. Samozřejmě je možné, že bude označeno více i méně částí a také je možné, že bude označeno částí 6, ale označení nebude správné.

### 7.1.3 Použitá nastavení

Program bude spuštěn nad výše popsanými daty, za použití různých nastavení viz tabulka 1. Jednotlivé parametry jsou popsány v podkapitole 6.3. Více nastavení je vybráno k nalezení a ukázce závislostí mezi jednotlivými parametry.

Například parametr leaf pozitivně ovlivní přesnost označení částí těla, ale naopak negativně rychlost výpočtu. Nastavení číslo jedna, dvě a tři by měly být rychlejší než nastavení číslo čtyři, protože mračno při výpočtu bude mít více bodů z důvodu, že je parametr leaf menší než u ostatních nastavení. Navíc je v nastavení č. 4 hledáno osm sousedních bodů, což je více než v nastavení č.1 a č.2. Z důvodu, že je bodů více, je také nutné použít menší hodnotu parametru persistence. Nastavení č.1, č.2 a č.3 se liší počtem hledaných sousedních bodů, cílem bude zjistit zda tento parametr zásadně ovlivňuje výsledek. Následně budou v podkapitole 8 ukázány jak jednotlivé

| | Nastavení č.1 | Nastavení č.2 | Nastavení č.3 | Nastavení č.4 |
|---|---|---|---|---|
| **Limit** | 1.8/2.3 | 1.8/2.3 | 1.8/2.3 | 1.8/2.3 |
| **Leaf** | 0.03 | 0.03 | 0.03 | 0.01 |
| **Radius** | 0.25 | 0.25 | 0.25 | 0.25 |
| **Persistence** | 0.008 | 0.008 | 0.008 | 0.005 |
| **Neighbors** | 6 | 4 | 8 | 8 |

Tabulka 1: Nastavení použitá pro experimenty

parametry ovlivňují výsledky, budou zvýrazněny klady a zápory jednotlivých nastavení a budou vyhodnoceny výše popsané hypotézy.

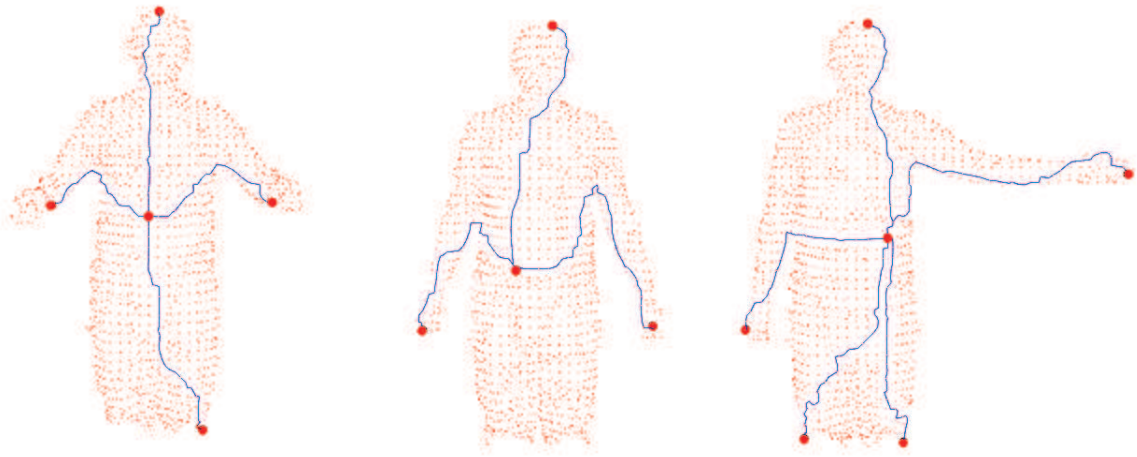
## 7.2 Provedení experimentů

Ke hromadnému spouštění experimentů jsem použil jednoduchý skript v jazyce *bash*, který postupně spouštěl program model_driver s jednotlivými snímky. Program byl spouštěn s přepínačem *experiments*, který zajistí automatické uložení snímku mračna. Skript poté změní příponu souboru na *.png*.

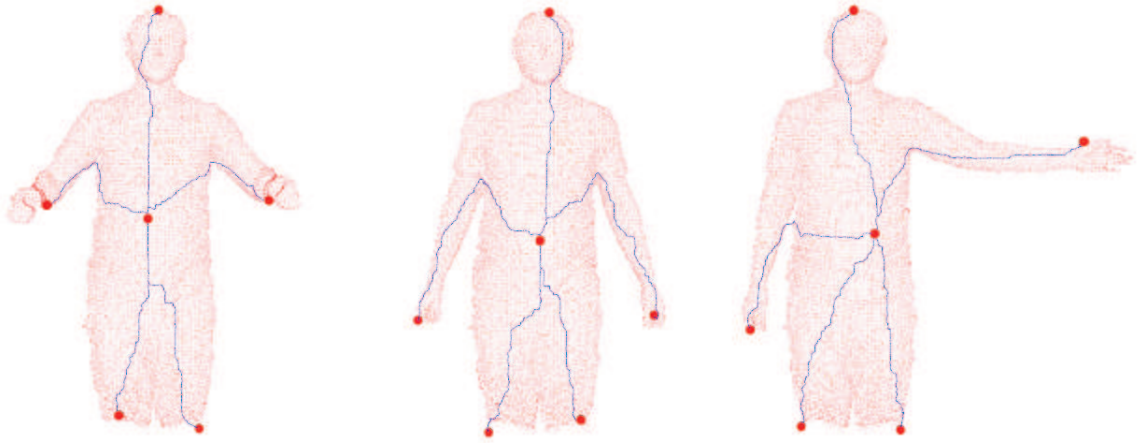
Během běhu programu je na standardní výstup vypsána cesta k mračnu, které je analyzováno, a také výsledný počet označení. Výstup programu model_driver byl zapisován do souboru *output.txt*. Tento soubor poté sloužil jako základ pro vypočtení metriky č.3, což je průměr počtu označených částí. Pro vypočtení doby zpracování jednoho snímku tj. metrika č.2, jsem využil Linuxové utility *time*, která mi pomohla změřit trvání zpracování každého mračna. Ze získaných časů jsem vypočetl průměr. K poslední metrice č.1 je implementována funkce, která načte ručně připravená data a z nich vypočte vzdálenosti mezi označenými body a body, které měly být doopravdy označeny v mračnu. Obsahem ručně připravených dat je název testované hloubkové mapy a indexy bodů v mračně, které se mají porovnávat (vypočtené programem a mnou ručně vybrané body).

# 8 Vyhodnocení

Jednotlivá nastavení byla nejdříve spuštěna na testovací množině č.1 a poté na testovací množině č.2 s parametrem *experiments* k získání snímků pro vizuální porovnání, a také pro získání vstupů pro vypočtení metrik. Protože vykreslování a ukládání snímků zaberou nějaký čas bylo nutné spustit experimenty i bez přepínače *display*. Byly spočteny všechny metriky jak bylo popsáno v předchozí kapitole č. 7. Vyzkoušel jsem obě metody hledání extrémů zmíněné v kapitole o implementaci. Zjistil jsem, že použití jedné nebo druhé varianty nemá výrazný vliv na celkové výsledky.



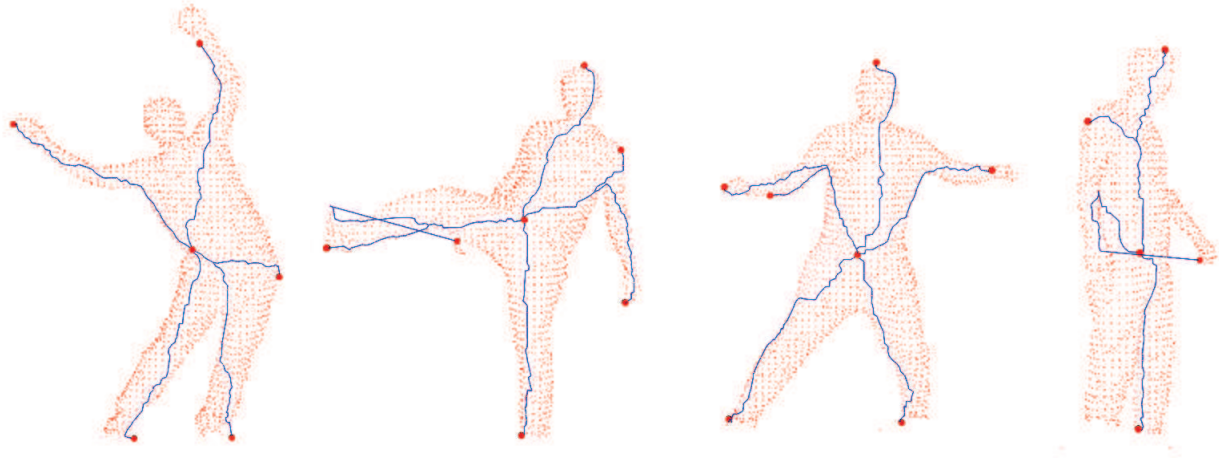
Obrázek 18: Ukázka výsledků s nastavením č.1 z testovací sady č.1



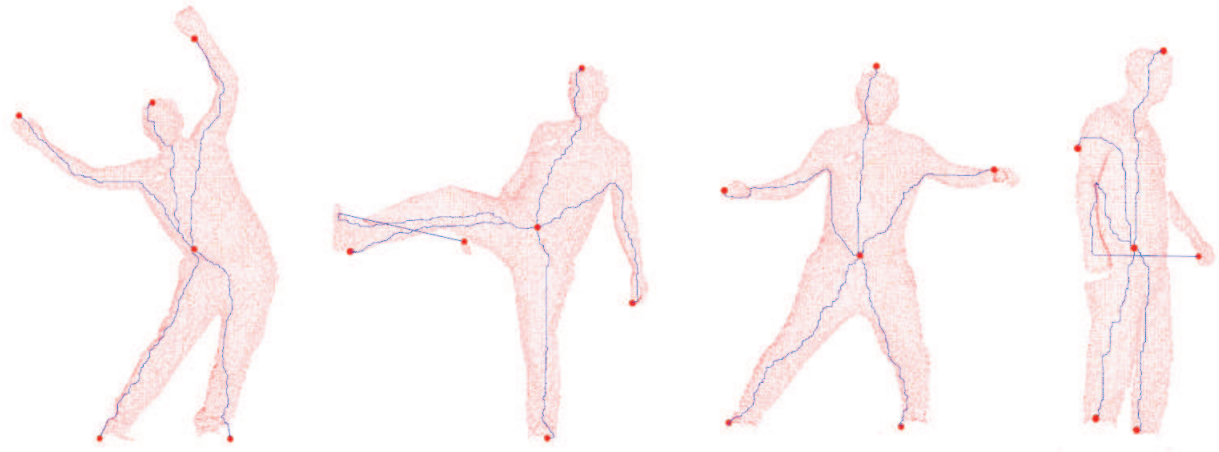
Obrázek 19: Ukázka výsledků s nastavením č.4 z testovací sady č.1

Na obrázcích 18 a 19 jsou ukázány výsledné analýzy některých mračen z testovací sady č.1. Navzájem jsou vždy porovnány varianty nastavení č.1 a č.4, mezi kterými by měl být vidět největší rozdíl v přesnosti označení částí těla.

Jak lze vidět použité nastavení č.4 podává přesnější výsledky než nastavení č.1. S nastavením č.1 jsou většinou označeny stejné oblasti jako s nastavením č.4, pokud jsou nalezeny. Některé ale nalezeny nejsou, zatímco s nastavením č.4 nalezeny jsou. Například na obrázku 18 nejsou nalezeny oblasti dolních končetin, kdežto na obrázku 19 nalezeny jsou. Také si lze všimnout, že centrální bod je s přesnějším nastavením (č.4) více uprostřed těla než s méně přesnějším nastavením (č.1).



Obrázek 20: Ukázka výsledků s nastavením č.1 z testovací sady č.2



Obrázek 21: Ukázka výsledků s nastavením č.4 z testovací sady č.2

Testovací sada č.2 nám dává mnohem lepší pohled na to, jaké jsou další rozdíly ve výše zmíněných nastaveních. Pro ukázku rozdílů nahlédněte do obrázků 20 a 21.

Na uvedených obrázcích lze vidět, jak se analýza mračen chová při odlišných pózách. Méně přesná varianta například neoznačila oblast hlavy na osobě první zleva. Naopak na osobě druhé z prava, označil program v méně přesné variantě o jednu část navíc. Lze také vidět, jak funguje program za situace, kdy je například jedna ruka za nohou, nebo při otočení osoby kolem své osy. Je nutné zmínit, že jsem zaznamenal i případy, kdy méně přesná varianta označila určitou část těla správně a přesnější neoznačila tuto část těla vůbec. Většinou se jednalo o případy, kdy v přesnější variantě byly vybrány body, které jsou blíže u sebe a následně byl jeden z nich eliminován, kdežto v méně přesné variantě jsou body dál od sebe a již nebyli v dosahu při eliminaci cest. Přesnějším nastavením parametru *radius*, by tyto případy nenastaly. Nad testovacími daty jsem však v globálním měřítku shledal nastavení č.4 jako přesnější.

V následujících podkapitolách bych rád shrnul jednotlivé výsledky výpočtů metrik a porovnal je mezi sebou a pokusil se prezentovat význam těchto výsledků.

## 8.1 Přesnost výsledků

Analýza naměřených hodnot nám dává určitý obraz toho, jak dobře či špatně program funguje. V tabulkách č. 2 a č. 3 máme statistický souhrn pro dvě testovací množiny, týkající se metriky počtu označených částí v hloubkové mapě. Lze vidět, že nastavení č.1 až nastavení č.3 mají velice podobné hodnoty, naopak nastavení č.4 se od nich liší významněji.

Z výsledků vyplývá, že první tři nastavení průměrně označili přibližně 4 části těla. Mnohem přesnější ale je hodnota váženého průměru, kde váhou je počet stejných hodnot v souboru hodnot. Vážený průměr se blíží hodnotě 5 v rámci všech nastavení. V případě nastavení č.4 dosahuje vážený průměr hodnoty menší o přibližně 0.2 vhledem k ostatním nastavením. Nastavení č.4 má však nejmenší rozptyl a směrodatnou odchylku, což znamená menší oscilaci hodnot a stabilnější výsledky.

| | Nastavení č.1 | Nastavení č.2 | Nastavení č.3 | Nastavení č.4 |
|---|---|---|---|---|
| Průměr | 3.87 | 3.93 | 3.85 | 3,47 |
| Vážený průměr | 3.87 | 3.93 | 3.85 | 3,47 |
| Medián | 4 | 4 | 4 | 4 |
| Rozptyl | 0,93 | 1,01 | 0,98 | 0,9 |
| Směrodatná odchylka | 0,97 | 1,01 | 0,99 | 0,95 |
| Šikmost | -0,09 | -0,11 | -0,27 | -0,29 |
| Špičatost | 4,28 | 3,95 | 4,22 | 3,23 |

Tabulka 2: Statistický souhrn pro testovací sadu č.1

Záporná šikmost nám říká, že většina hodnot se nachází nad průměrem napříč nastaveními. Pro nastavení č.4 doshuje nejmenší hodnoty.

Kladná špičatost nám říká, že rozdělení hodnot je špičatější než normální rozdělení. Nastavení č.4 se tak nejvíce přibližuje normálnímu rozdělení z daných nastavení, jelikož má hodnotu špičatosti neblíže 0 ze všech uvedených nastevení.

| | Nastavení č.1 | Nastavení č.2 | Nastavení č.3 | Nastavení č.4 |
|---|---|---|---|---|
| Průměr | 3,87 | 3,93 | 3,85 | 3,47 |
| Vážený průměr | 4.92 | 4.97 | 4.95 | 4.74 |
| Medián | 4 | 4 | 4 | 4 |
| Rozptyl | 0,93 | 1,01 | 0,98 | 0,9 |
| Směrodatná odchylka | 0,97 | 1,01 | 0,99 | 0,95 |
| Šikmost | -0,09 | -0,11 | -0,27 | -0,29 |
| Špičatost | 4,28 | 3,95 | 4,22 | 3,23 |

Tabulka 3: Statistický souhrn pro testovací sadu č.2

Naměřené časy jednoznačně říkají, že se zvyšujícím se počtem bodů, narůstá také časová náročnost na zpracování jednoho snímku. Při méně přesném nastavení je program schopen analyzovat jeden snímek za asi 250 milisekund, to jsou čtyři snímky za vteřinu. V přesnější variantě se pak dostáváme na přibližně dva snímky za tři sekundy.

| | Průměrný čas výpočtu [ms] s nastavením č.1 | Průměrný čas výpočtu [ms] s nastavením č.2 | Průměrný čas výpočtu [ms] s nastavením č.3 | Průměrný čas výpočtu [ms] s nastavením č.4 |
|---|---|---|---|---|
| Testovací sada č.1 | 313 | 255 | 235 | 721 |
| Testovací sada č.2 | 242 | 201 | 202 | 740 |

Tabulka 4: Průměrný výpočetní čas v závisloti na daném nastavení v milisekundách.

Změřené odchylky nám dávají přesnost na vybraných snímcích. Záměrně byly vybrány snímky, na kterých byly označeny všechny definované oblasti. Naměřené hodnoty nám dávají celkovou odchylku 6.7 centimetrů. Podíváme-li se na jednotlivé oblasti, je nejméně přesně zaměřená pravá ruka, což v dané množině bylo způsobeno jedním velmi špatným zaměřením v mračnu. Druhá nejhorší oblast je hlava, kde program zaměřuje z velké části temeno hlavy. Jako referenční bod jsem ale zvolil oblast, ve které se nachází nos, tedy prostředek obličeje. Vzniká tak průměrná odchylka 8.7 centimetrů.

| Odchylka oblast hlavy [mm] | Odchylka oblast pravá ruka [mm] | Odchylka oblast levá ruka [mm] | Odchylka oblast pravá noha [mm] | Odchylka oblast levá noha [mm] |
|---|---|---|---|---|
| 84 | 66 | 69 | 21 | 36 |
| 54 | 60 | 63 | 42 | 30 |
| 87 | 75 | 60 | 72 | 192 |
| 84 | 78 | 45 | 45 | 63 |
| 93 | 414 | 18 | 24 | 24 |
| 126 | 48 | 45 | 48 | 45 |
| 75 | 84 | 222 | 27 | 12 |
| 90 | 75 | 36 | 33 | 18 |
| 87 | 27 | 45 | 45 | 45 |
| 90 | 3 | 63 | 39 | 33 |
| **87** | **93** | **67** | **40** | **50** |
| **Celkový průměr** | | **67** | | |

Tabulka 5: Odchylky mezi vypočteným a ideálním označení částí těla v milimetrech.

## 8.2 Porovnání výsledků

K porovnání mám k dispozici dvě práce zmíněné v kapitole 4.2 a kapitole 4.1. Výsledky programu model_driver v porovnání s těmito publikacemi podává celkově horší výsledky. Pomocí rozhodovacích stromů bylo možné zpracovávat 200 snímků za sekundu na GPU a 50 snímků na CPU. Za pomocí geodetických vzdáleností a registraci mračen je možné zpracovávat přibližně 20 snímků za vteřinu. Oproti tomu program model_driver je schopen zpracovávat asi 4 snímky za sekundu.

Odchylka při označování jednotlivých oblastí je dosažena v těchto publikacích je také nižší než s programem model_driver a to řádově v jednotkách centimetrů. Například pro chodidla je rozdíl přibližně 2 centimetry, pro hlavu však 8 centimetrů. Navíc tyto publikace označují více oblastí než vytvořený program.

## 8.3 Možná vylepšení

Po porovnání s existujícími publikacemi je jisté, že je co vylepšovat. Během řešení této práce mě napadla řada vylepšení, které jsem již nestihl implementovat, nebo jsem se zrovna rozhodl pro jinou mi bližší variantu některého z dostupných řešení.

Rád bych zde zmínil pár vylepšení, které by program jednoznačně zlepšili. Dále bych chtěl zmínit pár nápadů, které by mohly vylepšit nějakým způsobem chod aplikace.

1. Urychlení hledání centrálního bodu. Tato část by šla urychlit dvěma způsoby. Prvním je si zaznamenávat již vypočtené vzdálenosti mezi jednotlivými body. Tyto se pak nebudou počítat tolikrát a urychlí se tím výpočet. Druhým řešením je provedení výpočtů na GPU.

2. Přesnost označování jednotlivých částí. Pro tento bod mě také napadlo více řešení. Jelikož program hodně závisí na použitých parametrech. Pokud jsou nastaveny vhodně, dosahuje program dobrých výsledků. Proto si myslím, že by se dalo některé kroky a nastavení parametrů počítat adaptivně, v závisloti na dané hloubkové mapě. Druhým možným řešením by bylo zapojení znalosti o fyziologii člověka a hledání správných cest za pomocí předpokládaných směrů a umístění cílových bodů cest. Například víme, že hlava bude vždy ve vrchní části obrázku apod. Dále jsem již v textu zmínil jeden nápad viz podkapitola 5.6.

3. Urychlení programu. K urychlení bych opět použil GPU, jelikož se hodně operací provádí přes knihovnu PCL, která má podporu pro zpracování mračen pomocí technologie CUDA.

4. Nahrazení některých částí programu rychejšími algoritmy, například eliminace cest.

# 9 Závěr

Tato diplomová práce se zabývala tvořením modelu osoby za pomocí analýzy hloubkových map, získaných pomocí senzoru Microsoft Kinect druhé generace. Cílem práce bylo vytvořit program, který na svém vstupu přijme hloubkovou mapu a následně jí analyzuje. Na výstupu se očekává určitý model osoby (řidiče) obsahující označení jednotlivých částí těla (hlava, dolní a horní končetiny). Nejdříve byl proveden návrh takové aplikace, který zohledňoval různá úskalí řešeného problému. Následně byl tento program implementován v programovacím jazyce *C++*. V rámci implementace byly použity známé algoritmy např. *Shortest-path first*, který hledá nejkratší cesty z jednoho vrcholu do ostatních vrcholů v grafu, ale také mnou navržené algoritmy např. *Eliminace cest*. Pro práci s mračny (hloubkovými mapami) jsem používal knihovnu PCL.

Rovněž byly navrženy experimenty sloužící k řádnému otestování aplikace. V tomto návrhu byly definovány různé metriky pro porovnání s existujícími metodami. Pro experimenty jsem připravil dvě testovací sady hloubkových map a provedl experimenty dle návrhu. Po provedení experimentů byly vyhodnoceny výsledky těchto experimentů. V porovnání s existujícími řešeními, nevykazuje program lepší výsledky, přesto však nejsou špatné. Aplikace není úplně přesná v situacích, kdy dochází k překryvům, nebo osoba zaujímá obecně složitější pózy. V základních pózách naopak podává dobré výsledky. Jednou z výhod aplikace je její konfigurovatelnost a možnost přizpůsobení nastavení dle vstupních dat.

Na základě experimentů a zkušeností nabytých při řešení této práce jsem zmínil také řadu vylepšení a doporučení v kapitole 8.3. Velice zajímavé mi přijde urychlení mnoha výpočtů na grafické kartě pomocí technologie CUDA, jelikož i použitá knihovna PCL má podporu pro práci s touto technologií.

# Literatura


[0] SHOTTON, Jamie, Toby SHARP, Alex KIPMAN, Andrew FITZGIBBON, Mark FINOCCHIO, Andrew BLAKE, Mat COOK a Richard MOORE. *Real-time human pose recognition in parts from single depth images*. Communications of the ACM [online]. 2013, 56(1), 116- [cit. 2016-04-03]. DOI: 10.1145/2398356.2398381. ISSN 00010782. Dostupné z: `http://dl.acm.org/citation.cfm?doid=2398356.2398381`.

[1] HANDRICH, Sebastian a Ayoub AL-HAMADI. *Full-Body Human Pose Estimation by Combining Geodesic Distances and 3D-Point Cloud Registration* [online]. s. 287 [cit. 2016-04-03]. DOI: 10.1007/978-3-319-25903-1_25. Dostupné z: `http://link.springer.com/10.1007/978-3-319-25903-1_25`.

[2] SIEK, Jeremy, Lie-Quan LEE a Andrew LUMSDAINE. *The boost graph library: user guide and reference manual*. Boston: Addison-Wesley, c2002. ISBN 0201729148.

[3] *Extracting and Filtering Minima and Maxima of 1D Functions* [online]. Saarbrücken: Weinkauf, 2013 [cit. 2016-04-03]. Dostupné z: `http://people.mpi-inf.mpg.de/~weinkauf/notes/persistence1d.html`.

[4] Coding4Fun Kinect Projects: Kinect 1 vs. Kinect 2, a quick side-by-side reference. *Channel 9: Videos for developers from the people building Microsoft Products and Services* [online]. Microsoft, 2016 [cit. 2016-04-03]. Dostupné z: `https://channel9.msdn.com/coding4fun/kinect/Kinect-1-vs-Kinect-2-a-side-by-side-reference`.

[5] Morphology: Distance Transform. *HIPR* [online]. Edinburgh: R. Fisher, S. Perkins, A. Walker and E. Wolfart, 2004 [cit. 2016-04-03]. Dostupné z: `http://homepages.inf.ed.ac.uk/rbf/HIPR2/distance.htm`.

[6] A 3-dimensional k-d tree [obrázek]. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2016 [cit. 2016-04-03]. Dostupné z: `https://en.wikipedia.org/wiki/K-d_tree`.

[7] Problém nejkratší cesty. *Algoritmus* [online]. Brno: INFO WEB, 2015 [cit. 2016-04-03]. Dostupné z: `https://www.algoritmy.net/article/36597/Nejkratsi-cesta`.

[8] CMake. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2016 [cit. 2016-04-03]. Dostupné z: `https://cs.wikipedia.org/wiki/CMake`.

[9] KULHAN, Jakub, ŠIMEČEK, Martin (ed.). CMake: tutoriál. In: *Programujte.com* [online]. Brno: devtea.cz, 2009 [cit. 2016-04-03]. ISSN 1801-1586. Dostupné z: `http://programujte.com/clanek/2009032800-cmake-tutorial/`.

[10] A Quick Tour of the Boost Graph Library. *Boost C++ Libraries* [online]. Indiana, 2015 [cit. 2016-04-03]. Dostupné z: http://www.boost.org/doc/libs/1_60_0/libs/graph/doc/quick_tour.html.

[11] *PCL - Point Cloud Library (PCL)* [online]. 2014 [cit. 2016-04-03]. Dostupné z: http://pointclouds.org/.

[12] Class Template Reference: pcl::VoxelGrid<PointT>. *Point Cloud Library (PCL)* [online]. 2016 [cit. 2016-04-03]. Dostupné z: http://docs.pointclouds.org/trunk/classpcl_1_1_voxel_grid.html.

[13] Documentation: The OpenNI Grabber Framework in PCL. *Point Cloud Library (PCL)* [online]. 2016 [cit. 2016-04-03]. Dostupné z: http://pointclouds.org/documentation/tutorials/openni_grabber.php.

[14] ALEXANDROV, Sergey. *Semi-automatic point cloud segmentation* [online prezentace]. 2014 [cit. 2016-04-03]. Dostupné z: http://www.pointclouds.org/assets/uploads/PCL-IAS13_Segmentation.pdf.

[15] *The Lean Mean C++ Option Parser* [online]. La Jolla: Slashdot Media, 2016 [cit. 2016-04-03]. Dostupné z: http://optionparser.sourceforge.net/.

[16] GANAPATHI, Varun, Christian PLAGEMANN, Daphne KOLLER a Sebastian THRUN. Real time motion capture using a single time-of-flight camera. In: *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition* [online]. IEEE, 2010, s. 755-762 [cit. 2016-04-03]. DOI: 10.1109/CVPR.2010.5540141. ISBN 978-1-4244-6984-0. Dostupné z: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5540141

[17] WEBB, Jarrett a James ASHLEY. *Beginning Kinect programming with the Microsoft Kinect SDK*. New York: Apress, 2012. Expert's voice in Microsoft. ISBN 1430241047.

[18] MALIK, Aamir Saeed, Tae Sun CHOI a Humaira NISAR. *Depth map and 3D imaging applications: algorithms and technologies*. Hershey, PA: Information Science Reference, c2012. ISBN 9781613503287.

# Seznam příloh

# A Obsah DVD

- Zdrojové kódy aplikace.
- Demonstrační data
- Elektronická verze této zprávy

# B Nápověda programu model_driver

```
model_driver --help
POUŽITÍ: model_driver [volby]

Volby:
  --help         Vypíše nápovědu a ukončí se.
  --experiments  Zapíná mód experimentů.
  --stream       Vykresluje proud PCD souborů.
  --render       Vykreslí mračno specifikované přepínači --file a --dir.
  --color        Pokud je specifikován tento parametr použije se černé pozadí
                 a barevný model bodů místo bílého pozadí a červených bodů.
  --verbose      Zapíná ladící výpisy během běhu programu.
  --display      Zapíná vykreslení mračna pomocí vizualizéru.
  --skeleton     Spustí funkci compute_skeleton.
  --dir          Název adresáře s testovacími daty.
  --file         Název PCD souboru.
                 Očekává datový typ {STRING} (řetězec).
  --limit        Vykreslovací vzdálenost bodů. Za touto hranicí jsou zahozeny.
                 Očekává datový typ {FLOAT} (desetinné číslo).
  --persistence  Odchylka použitá při hledání lokálních maxim.
                 Očekává datový typ {FLOAT} (desetinné číslo).
  --radius       Velikost okolí, ve kterém se budou hledat konce cest k eliminaci.
                 Očekává datový typ {FLOAT} (desetinné číslo).
  --leaf         Velikost voxelu.
                 Očekává datový typ {FLOAT} (desetinné číslo).
  --neighbors    Počet nejbližších sousedních bodů k nalezení.
                 Očekává datový typ {INT} (celé číslo).

Examples:
  model_driver --dir test_data_3 --file test_pcd_150.pcd --skeleton --limit 2.3
--leaf 0.01 --radius 0.25 --persistence 0.005 --neighbors 8 --display

  model_driver --dir test_data_3 --file test_pcd_044.pcd --render   --limit 2.3
--leaf 0.01 --persistence 0.03 --color
```

# C Ovládání PCL vizualizéru

Vizualizér spuštěný pomocí parametru `--display` se dá ovládat následovně:

```
      p, P    : změna na bodovou reprezentaci
      w, W    : změna na drátovou reprezentaci
      s, S    : změna na povrchovou reprezentaci

      j, J    : uloží screenshot momentálního okna ve formátu PNG
      c, C    : zobrazí parametry kamery/okna
      f, F    : letový režim k bodu

      e, E    : opuštění interkačního módu
      q, Q    : zastavení a ukončení

       +/-    : zvětšení/zmenšení celkové velikosti bodů
 +/- [+ ALT]  : přiblížení/oddálení

      g, G    : zobrazení měřítka (zapnuto/vypnuto)

o, O          : přepínání mezi perspektivní a paralelní projekcí
r, R [+ ALT]  : resetování kamery
CTRL + s, S   : uložení parametrů kamery
CTRL + r, R   : obnovení parametrů kamery

ALT + s, S    : zapnutí stereo módu (zapnuto/vypnuto)
ALT + f, F    : přepínání mezi maximalizovaným oknem a původní velikostí okna
```